

Freescale XGATE Compiler

metrowerks

Metrowerks and the Metrowerks logo are registered trademarks of Metrowerks Corp. in the US. CodeWarrior is a trademark or registered trademark of Metrowerks Corp. in the US and/or other countries. All other tradenames and trademarks are the property of their respective owners.

Copyright © Metrowerks Corporation. 2004. ALL RIGHTS RESERVED.

The reproduction and use of this document and related materials are governed by a license agreement media, it may be printed for non-commercial personal use only, in accordance with the license agreement related to the product associated with the documentation. Consult that license agreement before use or reproduction of any portion of this document. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 800-377-5416 (if outside the US call +1 512-997-4700). Subject to the foregoing non-commercial personal use, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

Metrowerks reserves the right to make changes to any product described or referred to in this document without further notice. Metrowerks makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Metrowerks assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Metrowerks software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.**

USE OF ALL SOFTWARE, DOCUMENTATION AND RELATED MATERIALS ARE SUBJECT TO THE METROWERKS END USER LICENSE AGREEMENT FOR SUCH PRODUCT.

How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	http://www.metrowerks.com
Sales	Voice: 800-377-5416 Fax: 512-996-4910 Email: sales@metrowerks.com
Technical Support	Voice: 800-377-5416 Voice: 512-996-5300 Email: support@metrowerks.com

Table of Contents

1 Using the Compiler	37
Introduction	37
Application Programs	37
Startup Command Line Options	38
Literature	38
Highlights	40
Structure of this Document	40
Specific Configuration Markers	41
CodeWarrior Integration	41
Integration into Microsoft Visual Studio (Visual C++ V5.0 or later)	51
C++, EC++, compactC++	54
Object File Formats	55
Graphical User Interface	59
Launching a Tool	59
Tip of the Day	60
Main Window	61
Window Title	61
Content Area	62
Tool Bar	63
Status Bar	64
Menu Bar	64
Standard Types Dialog Box	78
Options Dialog Box	79
Compiler Smart Control Dialog Box	81
Message Settings Dialog Box	82
About Box	85
Specifying the Input File	85
Message/Error Feedback	86
Environment	89
The Current Directory	90
Environment Macros	91

Table of Contents

Global Initialization File (MCUTOOLS.INI) (PC only)	91
Local Configuration File (usually project.ini)	92
Paths	93
Line Continuation	94
Environment Variable Details	95
COMPOPTIONS: Default Compiler Options	96
COPYRIGHT: Copyright Entry in Object File	97
DEFAULTDIR: Default Current Directory	98
ENVIRONMENT: Environment File Specification	99
ERRORFILE: Error File Name Specification	100
GENPATH: #include “File” Path	102
INCLUDETIME: Creation Time in Object File	103
LIBRARYPATH: ‘include <File>’ Path	105
OBJPATH: Object File Path	107
TEXTPATH: Text File Path	108
TMP: Temporary Directory	109
USELIBPATH: Using LIBPATH Environment Variable	110
USERNAME: User Name in Object File	111
Files	113
Input Files.	113
Output Files	114
Compiler Options	117
Option Recommendation	118
Compiler Option Details	119
-!: File Names to DOS Length	123
-AddIncl: Additional Include File	125
-Ansi: Strict ANSI	126
-BfaB: Bit Field Byte Allocation	128
-BfaGapLimitBits: Bit Field Gap Limit	130
-BfaTSR: Bit Field Type Size Reduction.	132
-C++ (-C++f, -C++e, -C++c): C++ Support	134
-Cc: Allocate Constant Objects into ROM	136
-Ccx: Cosmic compatibility mode for space modifiers and interrupt handlers .	138

-Ci: Tri- and Bigraph Support	141
-Cn: Disable compactC++ features	145
-Cni: No Integral Promotion	147
-Cppc: C++ Comments in ANSI-C	150
-Cq: Propagate const and volatile Qualifiers for structs	152
-CsIni0: Assume SP register is zero initialized at thread start	154
-Cstv: Initialize Stack	156
-CswMaxLF: Maximum Load Factor for Switch Tables	157
-CswMinLB: Minimum Number of Labels for Switch Tables	159
-CswMinLF: Minimum Load Factor for Switch Tables	161
-CswMinSLB: Minimum Number of Labels for Search Switch Tables	163
-Cu: Loop Unrolling	165
-Cx: No Code Generation	168
-D: Macro Definition	169
-Ec: Conversion from 'const T*' to 'T*'	171
-Encrypt: Encrypt Files	173
-Ekey: Encryption Key	175
-Env: Set Environment Variable	177
-F (-Fh, -F1, -F1o, -F2, -F2o, -F6, -F7): Object File Format	179
-H: Short Help	182
-I: Include File Path	184
-La: Generate Assembler Include File	186
-Lasm: Generate Listing File	188
-Lasmc: Configure Listing File	190
-Ldf: Log Predefined Defines to File	192
-Li: List of Included Files	194
-Lic: License Information	196
-LicA: License Information about every Feature in Directory	197
-LicBorrow: Borrow License Feature	199
-LicWait: Wait until floating License is available from floating License Server .	201
-Ll: Statistics About Each Function	203
-Lm: List of Included Files in make Format	205
-LmCfg: Configuration of List of Included Files in make Format	207

Table of Contents

-Lo: Object File List	210
-Lp: Preprocessor Output	212
-LpCfg: Preprocessor Output configuration	214
-LpX: Stop After Preprocessor	216
-N: Display Notify Box	217
-NoBeep: No Beep in Case of an Error	219
-NoDebugInfo: Do Not Generate Debug Information	220
-NoEnv: Do Not Use Environment	222
-NoPath: Strip Path Info	224
-Oa: Alias Analysis Options	225
-O (-Os, -Ot): Main Optimization Target	227
-ObjN: Object File Name Specification	229
-Oc: Common Subexpression Elimination (CSE)	231
-Od: Disable mid-level Optimizations	233
-Odb: Disable mid-level Branch Optimizations	235
-OdocF: Dynamic Option Configuration for Functions	237
-Oi: Inlining	240
-Oilib: Inline Library Functions	242
-OnBRA: Disable JAL to BRA Optimization	245
-OnCopyDown: Do Generate Copy Down Information for Zero Values	248
-OnCstVar: Disable CONST Variable by Constant Replacement	250
-OnPMNC: Disable Code Generation for NULL Pointer to Member Check	252
-Ont: Disable Tree Optimizer	254
-Pe: Preprocessing Escape Sequences in Strings	261
-Pio: Include Files Only Once	263
-Prod: Specify Project File at Startup	265
-Qvtp: Qualifier for Virtual Table Pointers	267
-T: Flexible Type Management	269
-V: Prints the Compiler Version	276
-View: Application Standard Occurrence	278
-WErrFile: Create "err.log" Error File	280
-Wmsg8x3: Cut File Names in Microsoft Format to 8.3	282
-WmsgCE: RGB Color for Error Messages	284

-WmsgCF: RGB Color for Fatal Messages	285
-WmsgCI: RGB Color for Information Messages	286
-WmsgCU: RGB Color for User Messages	287
-WmsgCW: RGB Color for Warning Messages	288
-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode	289
-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode	292
-WmsgFob: Message Format for Batch Mode	294
-WmsgFoi: Message Format for Interactive Mode	296
-WmsgFonf: Message Format for no File Information	298
-WmsgFonp: Message Format for no Position Information	300
-WmsgNe: Number of Error Messages	302
-WmsgNi: Number of Information Messages	304
-WmsgNu: Disable User Messages	305
-WmsgNw: Number of Warning Messages	307
-WmsgSd: Setting a Message to Disable	308
-WmsgSe: Setting a Message to Error	310
-WmsgSi: Setting a Message to Information	312
-WmsgSw: Setting a Message to Warning	314
-WOutFile: Create Error Listing File	316
-Wpd: Error for Implicit Parameter Declaration	318
-WStdout: Write to Standard Output	320
-W1: No Information Messages	322
-W2: No Information and Warning Messages	323
Compiler Predefined Macros	325
Compiler Vendor Defines	325
Product Defines	326
Data Allocation Defines	326
Various Defines for Compiler Option Settings	327
Option Checking in C Code	327
ANSI-C Standard Types 'size_t', 'wchar_t' and 'ptrdiff_t' Defines	328
Division and Modulus	331
Object File Format Defines	331
Bit Field Defines	332

Table of Contents

Type Information Defines	335
Freescale XGATE Specific Defines	337
Compiler Pragmas	339
Pragma Details	339
#pragma CODE_SEG: Code Segment Definition	341
#pragma CONST_SEG: Constant Data Segment Definition	344
#pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing	347
#pragma DATA_SEG: Data Segment Definition	348
#pragma INLINE: Inline Next Function Definition	351
#pragma INTO_ROM: Put Next Variable Definition into ROM	352
#pragma LINK_INFO: Pass Information to the Linker	354
#pragma LOOP_UNROLL: Force Loop Unrolling	356
#pragma mark: Entry in CodeWarrior IDE Function List	357
#pragma MESSAGE: Message Setting	359
#pragma NO_ENTRY: No Entry Code	361
#pragma NO_EXIT: No Exit Code	363
#pragma NO_FRAME: No Frame Code	365
#pragma NO_INLINE: Do not Inline next Function Definition	367
#pragma NO_LOOP_UNROLL: Disable Loop Unrolling	369
#pragma NO_RETURN: No Return Instruction	370
#pragma NO_STRING_CONSTR: No String Concatenation During Preprocessing	
372	
#pragma ONCE: Include Once	373
#pragma OPTION: Additional Options	374
#pragma pop: restore pragma state	377
#pragma push: save pragma state	379
#pragma REALLOC_OBJ: Object Reallocation	381
#pragma STRING_SEG: String Segment Definition	383
#pragma TEST_CODE: Check Generated Code	385
#pragma TRAP_PROC: Mark Function as Interrupt Function	387
ANSI-C Front End	389
Implementation Features	389
ANSI-C Standard	407

Floating Type Formats	410
Volatile Objects, Absolute Variables	414
Bit Fields	414
Segmentation	415
Optimizations	419
Using Qualifiers for Pointers	423
Defining C Macros Containing HLI Assembler Code	425
C++ Front End	437
Overview	437
Implementation	437
Features	437
Additional Keywords in C++	439
C Linkage	439
The Standard Predefined Macro <code>_cplusplus</code>	440
Special Segment Support for C++	440
Differences between ANSI-C and C++	441
C++ and Embedded Systems	443
Short C++ Tutorial	450
C++ Name Encoding, Type-safe Linkage	492
Generating Compact Code	497
Compiler Options	497
<code>_SHORT_SEG</code> Segments	497
Defining IO Registers	499
Programming Guidelines	500
Freescale XGATE Back End	507
Non-ANSI Keywords	507
Data Types	507
Register Usage	509
Call Protocol and Calling Conventions	510
Stack Frames	511
Pragmas	514

Table of Contents

Interrupt Functions	514
Intrinsics	515
Segmentation	518
Optimizations	519
Programming Hints	520
High Level Inline Assembler for Freescale XGATE	521
Syntax	521
Special Features	523
2 Messages	527
Message Kinds	527
Message Details	527
Message List	528
C1: Unknown message occurred	529
C2: Message overflow, skipping <kind> messages	529
C50: Input file '<file>' not found	529
C51: Cannot open statistic log file <file>	530
C52: Error in command line <cmd>	530
C64: Line Continuation occurred in <FileName>	530
C65: Environment macro expansion message '<description>' for <variablename>	
531	
C66: Search path <Name> does not exist	532
C1000: Illegal identifier list in declaration	532
C1001: Multiple const declaration makes no sense	533
C1002: Multiple volatile declaration makes no sense	533
C1003: Illegal combination of qualifiers	534
C1004: Redefinition of storage class	534
C1005: Illegal storage class	534
C1006: Illegal storage class	535
C1007: Type specifier mismatch	535
C1008: Typedef name expected	536
C1009: Invalid redeclaration	536
C1010: Illegal enum redeclaration	537

C1012: Illegal local function definition	537
C1013: Old style declaration	538
C1014: Integral type expected or enum value out of range	539
C1015: Type is being defined	539
C1016: Parameter redeclaration not permitted	540
C1017: Empty declaration	540
C1018: Illegal type composition	541
C1019: Incompatible type to previous declaration	541
C1020: Incompatible type to previous declaration	542
C1021: Bit field type is not 'int'	542
C1022: 'far' used in illegal context	543
C1023: 'near' used in illegal context	543
C1024: Illegal bit field width	544
C1025: ',' expected before '...'	544
C1026: Constant must be initialized	545
C1027: Reference must be initialized	545
C1028: Member functions cannot be initialized	546
C1029: Undefined class	546
C1030: Pointer to reference illegal	546
C1031: Reference to reference illegal	547
C1032: Invalid argument expression	547
C1033: Ident should be base class or data member	548
C1034: Unknown kind of linkage	548
C1035: Friend must be declared in class declaration	549
C1036: Static member functions cannot be virtual	549
C1037: Illegal initialization for extern variable in block scope	550
C1038: Cannot be friend of myself	550
C1039: Typedef-name or ClassName expected	551
C1040: No valid :: classname specified	551
C1041: Multiple access specifiers illegal.	552
C1042: Multiple virtual declaration makes no sense	552
C1043: Base class already declared in base list	553
C1044: User defined Constructor is required	553

Table of Contents

C1045: <Special member function> not generated	554
C1046: Cannot create compiler generated <Special member function> for nameless class	554
C1047: Local compiler generated <Special member function> not supported .	555
C1048: Generate compiler defined <Special member function>	555
C1049: Members cannot be extern	556
C1050: Friend must be a class or a function	556
C1051: Invalid function body	557
C1052: Unions cannot have class/struct object members containing Con/ Destructor/Assign-Operator	557
C1053: Nameless class cannot have member functions	558
C1054: Incomplete type or function in class/struct/union	558
C1055: External linkage for class members not possible	559
C1056: Friend specifier is illegal for data declarations	559
C1057: Wrong return type for <FunctionKind>	560
C1058: Return type for FunctionKind must be <ReturnType>	560
C1059: Parameter type for <FunctionKind> parameter <No> must be <Type> .	561
C1060: <FunctionKind> wrong number of parameters	561
C1061: Conversion operator must not have return type specified before operator keyword	561
C1062: Delete can only be global, if parameter is (void *)	562
C1063: Global or static-member operators must have a class as first parameter	562
C1064: Constructor must not have return type	563
C1065: 'inline' is the only legal storage class for Constructors	563
C1066: Destructor must not have return type	564
C1067: Object is missing decl specifiers	564
C1068: Illegal storage class for Destructor	565
C1069: Wrong use of far/near/rom/uni/paged in local scope	565
C1070: Object of incomplete type	566
C1071: Redefined extern to static	566
C1072: Redefined extern to static	567
C1073: Linkage specification contradicts earlier specification	567
C1074: Wrong member function definition	568
C1075: Typedef object id already used as tag	568

C1076: Illegal scope resolution in member declaration	569
C1077: <FunctionKind> must not have parameters	569
C1078: <FunctionKind> must be a function	570
C1080: Constructor/destructor: Parenthesis missing	570
C1081: Not a static member.	571
C1082: <FunctionKind> must be non-static member of a class/struct	571
C1084: Not a member	572
C1085: <ident> is not a member	572
C1086: Global unary operator must have one parameter.	572
C1087: Static unary operator must have one parameter	573
C1088: Unary operator must have no parameter	573
C1089: Global binary operator must have two parameters	574
C1090: Static binary operator must have two parameters	574
C1091: Binary operator must have one parameter	574
C1092: Global unary/binary operator must have one or two parameters	575
C1093: Static unary/binary operator must have one or two parameters	575
C1094: Unary/binary operator must have no or one parameter	575
C1095: Postfix ++/-- operator must have integer parameter	576
C1096: Illegal index value	576
C1097: Array bounds missing	576
C1098: Modifiers for non-member or static member functions illegal	577
C1099: Not a parameter type	577
C1100: Reference to void illegal	578
C1101: Reference to bitfield illegal	578
C1102: Array of reference illegal.	579
C1103: Second C linkage of overloaded function not allowed	579
C1104: Bit field type is neither integral nor enum type	580
C1105: Backend does not support non-int bitfields.	580
C1106: Non-standard bitfield type	581
C1107: Long long bit fields not supported yet.	581
C1108: Constructor cannot have own class/struct type as first and only parameter	
581	
C1109: Generate call to Copy Constructor	582

Table of Contents

C1110: Inline specifier is illegal for data declarations	583
C1111: Bitfield cannot have indirection	583
C1112: Interrupt specifier is illegal for data declaration	584
C1113: Interrupt specifier used twice for same function	584
C1114: Illegal interrupt number	584
C1115: Template declaration must be class or function	585
C1116: Template class needs a tag	585
C1117: Illegal template/non-template redeclaration	586
C1118: Only bases and class member functions can be virtual	586
C1119: Pure virtual function qualifier should be (=0)	587
C1120: Only virtual functions can be pure	587
C1121: Definition needed if called with explicit scope resolution	588
C1122: Cannot instantiate abstract class object	589
C1123: Cannot instantiate abstract class as argument type	590
C1124: Cannot instantiate abstract class as return type	591
C1125: Cannot instantiate abstract class as a type of explicit conversion	591
C1126: Abstract class cause inheriting pure virtual without overriding function(s) 592	
C1127: Constant void type probably makes no sense	593
C1128: Class contains private members only	593
C1129: Parameter list missing in pointer to member function type.	594
C1130: This C++ feature is disabled in your current cC++/EC++ configuration	595
C1131: Illegal use of global variable address modifier	595
C1132: Cannot define an anonymous type inside parentheses	596
C1133: Such an initialization requires STATIC CONST INTEGRAL member	597
C1134: Static data members are not allowed in local classes	598
C1135: Ignore Storage Class Specifier cause it only applies on objects	598
C1136: Class <Ident> is not a correct nested class of class <Ident>	599
C1137: Unknown or illegal segment name	600
C1138: Illegal segment type.	600
C1139: Interrupt routine should not have any return value nor any parameter	601
C1140: This function is already declared and has a different prototype	602
C1141: Ident <ident> cannot be allocated in global register	603

C1142: Invalid Cosmic modifier. Accepted: @near, @far, @tiny or @interrupt (-ANSI off)	603
C1143: Ambiguous Cosmic space modifier. Only one per declaration allowed .	604
C1144: Multiple restrict declaration makes no sense	605
C1390: Implicit virtual function	605
C1391: Pseudo Base Class is added to this class.	606
C1392: Pointer to virtual methods table not qualified for code address space (use -Qvtprm or -Qvtpuni)	608
C1393: Delta value does not fit into range (option -TvtD)	608
C1395: Classes should be the same or derive one from another	609
C1396: No pointer to STATIC member: use classic pointer	611
C1397: Kind of member and kind of pointer to member are not compatible .	612
C1398: Pointer to member offset does not fit into range of given type (option -Tpmo)	613
C1400: Missing parameter name in function head	614
C1401: This C++ feature has not been implemented yet.	614
C1402: This C++ feature (<Feature>) is not implemented yet	615
C1403: Out of memory	615
C1404: Return expected	616
C1405: Goto <undeclared Label> in this function	616
C1406: Illegal use of identifierList	617
C1407: Illegal function-redefinition	617
C1408: Incorrect function-definition	617
C1409: Illegal combination of parameterlist and identlist	618
C1410: Parameter-declaration - identifier-list mismatch.	618
C1411: Function-definition incompatible to previous declaration	618
C1412: Not a function call, address of a function	619
C1413: Illegal label-redeclaration	619
C1414: Casting to pointer of non base class	620
C1415: Type expected	620
C1416: No initializer can be specified for arrays.	621
C1417: Const/volatile not allowed for type of new operator	621
C1418:] expected for array delete operator.	622
C1419: Non-constant pointer expected for delete operator	622

Table of Contents

C1420: Result of function-call is ignored	623
C1421: Undefined class/struct/union	623
C1422: No default Ctor available	624
C1423: Constant member must be in initializer list.	625
C1424: Cannot specify explicit initializer for arrays	625
C1425: No Destructor available to call	626
C1426: Explicit Destructor call not allowed here	626
C1427: 'this' allowed in member functions only	626
C1428: No wide characters supported	627
C1429: Not a destructor id	627
C1430: No destructor in class/struct declaration	628
C1431: Wrong destructor call	628
C1432: No valid classname specified	629
C1433: Explicit Constructor call not allowed here	629
C1434: This C++ feature is not yet implemented	630
C1435: Return expected	630
C1436: delete needs number of elements of array	631
C1437: Member address expected	632
C1438: ... is not a pointer to member ident	633
C1439: Illegal pragma __OPTION_ACTIVE__, <Reason>	634
C1440: This is causing previous message <MsgNumber>	635
C1441: Constant expression shall be integral constant expression	635
C1442: Typedef cannot be used for function definition	636
C1443: Illegal wide character	636
C1444: Initialization of <Variable> is skipped by 'case' label	637
C1445: Initialization of <Variable> is skipped by 'default' label.	638
C1800: Implicit parameter-declaration (missing prototype) for '<FuncName>' .	639
C1801: Implicit parameter-declaration for '<FuncName>'	639
C1802: Must be static member	640
C1803: Illegal use of address of function compiled under the pragma REG_PROTOTYPE	641
C1804: Ident expected	641
C1805: Non standard conversion used.	641

C1806: Illegal cast-operation	642
C1807: No conversion to non-base class	642
C1808: Too many nested switch-statements	643
C1809: Integer value for switch-expression expected	644
C1810: Label outside of switch-statement	644
C1811: Default-label twice defined	645
C1812: Case-label-value already present.	645
C1813: Division by zero	645
C1814: Arithmetic or pointer-expression expected	646
C1815: <Name> not declared (or typename)	646
C1816: Unknown struct- or union-member.	647
C1817: Parameter cannot be converted to non-constant reference	648
C1819: Constructor call with wrong number of arguments.	648
C1820: Destructor call must have 'void' formal parameter list	649
C1821: Wrong number of arguments	649
C1822: Type mismatch.	650
C1823: Undefining an implicit parameter-declaration.	650
C1824: Indirection to different types	651
C1825: Indirection to different types	652
C1826: Integer-expression expected	652
C1827: Arithmetic types expected	653
C1828: Illegal pointer-subtraction	653
C1829: + - incompatible Types	654
C1830: Modifiable lvalue expected	654
C1831: Wrong type or not an lvalue	655
C1832: Const object cannot get incremented	655
C1833: Cannot take address of this object	656
C1834: Indirection applied to non-pointer	656
C1835: Arithmetic operand expected	657
C1836: Integer-operand expected	657
C1837: Arithmetic type or pointer expected	658
C1838: Unknown object-size: sizeof (incomplete type)	658
C1839: Variable of type struct or union expected	658

Table of Contents

C1840: Pointer to struct or union expected	659
C1842: [incompatible types	659
C1843: Switch-expression: integer required	660
C1844: Call-operator applied to non-function.	660
C1845: Constant integer-value expected	660
C1846: Continue outside of iteration-statement	661
C1847: Break outside of switch or iteration-statement	661
C1848: Return <expression> expected	662
C1849: Result returned in void-result-function	662
C1850: Incompatible pointer operands	663
C1851: Incompatible types	663
C1852: Illegal sizeof operand	664
C1853: Unary minus operator applied to unsigned type	664
C1854: Returning address of local variable.	665
C1855: Recursive function call	665
C1856: Return <expression> expected	666
C1857: Access out of range	667
C1858: Partial implicit parameter-declaration.	667
C1859: Indirection operator is illegal on Pointer To Member operands	668
C1860: Pointer conversion: possible loss of data	669
C1861: Illegal use of type ‘void’	669
C2000: No constructor available	670
C2001: Illegal type assigned to reference.	670
C2004: Non-volatile reference initialization with volatile illegal	671
C2005: Non-constant reference initialization with constant illegal.	671
C2006: (un)signed char reference must be const for init with char.	672
C2007: Cannot create temporary for reference in class/struct.	672
C2008: Too many arguments for member initialization	673
C2009: No call target found!	673
C2010: <Name> is ambiguous.	674
C2011: <Name> can not be accessed	675
C2012: Only exact match allowed yet or ambiguous!.	676
C2013: No access to special member of base class	676

C2014: No access to special member of member class	677
C2015: Template is used with the wrong number of arguments	678
C2016: Wrong type of template argument	678
C2017: Use of incomplete template class	679
C2018: Generate class/struct from template	679
C2019: Generate function from template.	680
C2020: Template parameter not used in function parameter list	680
C2021: Generate NULL-check for class pointer	681
C2022: Pure virtual can be called only using explicit scope resolution	682
C2023: Missing default parameter	683
C2024: Overloaded operators cannot have default arguments.	683
C2025: Default argument expression can only contain static or global objects or constants	684
C2200: Reference object type must be const	684
C2201: Initializers have too many dimensions	685
C2202: Too many initializers for global Ctor arguments.	685
C2203: Too many initializers for Ctor arguments	686
C2204: Illegal reinitialization	687
C2205: Incomplete struct/union, object can not be initialized.	688
C2206: Illegal initialization of aggregate type.	688
C2207: Initializer must be constant	689
C2209: Illegal reference initialization	690
C2210: Illegal initialization of non-aggregate type	690
C2211: Initialization of a function	691
C2401: Pragma <ident> expected	691
C2402: Variable <Ident> <State>.	692
C2450: Expected: <list of expected keywords and tokens>	692
C2550: Too many nested scopes	693
C2700: Too many numbers	694
C2701: Illegal floating-point number	694
C2702: Number too large for float	695
C2703: Illegal character in float number	695
C2704: Illegal number	696

Table of Contents

C2705: Possible loss of data	696
C2706: Octal Number	697
C2707: Number too large	698
C2708: Illegal digit	698
C2709: Illegal floating-point exponent ('-', '+' or digit expected)	699
C2800: Illegal operator	699
C2801: <Symbol> missing"	700
C2802: Illegal character found: <Character>	700
C2803: Limitation: Parser was going out of sync!	701
C2900: Constant condition found, removing loop	701
C2901: Unrolling loop	702
C3000: File-stack-overflow (recursive include?)	702
C3100: Flag stack overflow -- flag ignored	703
C3200: Source file too big	704
C3201: Carriage-Return without a Line-Feed was detected	704
C3202: Ident too long	705
C3300: String buffer overflow	705
C3301: Catenated string too long	706
C3302: Prae_number-buffer overflow	706
C3303: Implicit concatenation of strings	707
C3304: Too many internal ids, split up compilation unit.	708
C3400: Cannot initialize object (dest too small)	708
C3401: Resulting string is not zero terminated	709
C3500: Not supported fixup-type for ELF-Output occurred	709
C3501: ELF Error <Description>	710
C3600: Function has no code: remove it!	710
C3601: Pragma TEST_CODE: mode <Mode>, size given <Size> expected <Size>, hashcode given <HashCode>, expected <HashCode>	711
C3602: Global objects: <Number>, Data Size (RAM): <Size>, Const Data Size (ROM): <Size>	711
C3603: Static '<Function>' was not defined	712
C3604: Static '<Object>' was not referenced	712
C3605: Runtime object '<Object>' is used at PC <PC>	713

C3606: Initializing object '<Object>'	713
C3700: Special opcode too large	714
C3701: Too many attributes for DWARF2.0 Output	714
C3800: Segment name already used	715
C3801: Segment already used with different attributes	715
C3802: Segment pragma incorrect	716
C3803: Illegal Segment Attribute	716
C3804: Predefined segment '<segmentName>' used	717
C3900: Return value too large	717
C4000: Condition always is TRUE	718
C4001: Condition always is FALSE	718
C4002: Result not used	719
C4003: Shift count converted to unsigned char	719
C4004: BitSet/BitClr bit number converted to unsigned char	720
C4005: Expression with a cast is not a lvalue	721
C4006: Expression too complex	721
C4100: Converted bit field signed -1 to 1 in comparison	722
C4101: Address of bitfield is illegal	723
C4200: Other segment than in previous declaration	723
C4201: pragma <name> was not handled	724
C4202: Invalid pragma OPTION, <description>	724
C4203: Invalid pragma MESSAGE, <description>	725
C4204: Invalid pragma REALLOC_OBJ, <description>	726
C4205: Invalid pragma LINK_INFO, <description>	726
C4300: Call of an empty function removed	727
C4301: Inline expansion done for function call	727
C4302: Could not generate inline expansion for this function call	728
C4303: Illegal pragma <name>	729
C4400: Comment not closed	729
C4401: Recursive comments not allowed	730
C4402: Redefinition of existing macro '<MacroName>'	730
C4403: Macro-buffer overflow	731
C4404: Macro parents not closed	731

Table of Contents

C4405: Include-directive followed by illegal symbol	732
C4406: Closing '>' missing	732
C4407: Illegal character in string or closing '>' missing	732
C4408: Filename too long	733
C4409: a ## b: the concatenation of a and b is not a legal symbol	733
C4410: Unbalanced Parentheses	734
C4411: Maximum number of arguments for macro expansion reached	734
C4412: Maximum macro expansion level reached	735
C4413: Assertion: pos failed	736
C4414: Argument of macro expected	736
C4415: ')' expected	737
C4416: Comma expected	737
C4417: Mismatch number of formal, number of actual parameters	737
C4418: Illegal escape sequence	738
C4419: Closing “ missing	739
C4420: Illegal character in string or closing " missing	739
C4421: String too long	740
C4422: ' missing	740
C4423: Number too long	741
C4424: # in substitution list must be followed by name of formal parameter	741
C4425: ## in substitution list must be preceded and followed by a symbol	742
C4426: Macro must be a name.	742
C4427: Parameter name expected	743
C4428: Maximum macro arguments for declaration reached	743
C4429: Macro name expected	744
C4430: Include macro does not expand to string.	744
C4431: Include "filename" expected	745
C4432: Macro expects ‘(.	745
C4433: Defined <name> expected	746
C4434: Closing ')' missing	746
C4435: Illegal expression in conditional expression	746
C4436: Name expected	747
C4437: Error-directive found: <message>	747

C4438: Endif-directive missing	748
C4439: Source file <file> not found	748
C4440: Unknown directive: <directive>	749
C4441: Preprocessor output file <file> could not be opened	749
C4442: Endif-directive missing	750
C4443: Undefined Macro 'MacroName' is taken as 0	750
C4444: Line number for #line directive must be > 0 and <= 32767	751
C4445: Line number for #line directive expected	752
C4446: Missing macro argument(s)	752
C4447: Unexpected tokens following preprocessor directive - expected a newline 753	
C4448: Warning-directive found: <message>	754
C4449: Exceeded preprocessor #if level of 4092	754
C4700: Illegal pragma TEST_ERROR	755
C4701: pragma TEST_ERROR: Message <ErrorNumber> did not occur	755
C4800: Implicit cast in assignment	755
C4801: Too many initializers	756
C4802: String-initializer too large	756
C4900: Function differs in return type only.	756
C5000: Following condition fails: sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)	757
C5001: Following condition fails: sizeof(float) <= sizeof(double) <= sizeof(long double) <= sizeof(long long double)	757
C5002: Illegal type	758
C5003: Unknown array-size	758
C5004: Unknown struct-union-size	759
C5005: PACE illegal type	759
C5006: Illegal type settings for HIWARE Object File Format	759
C5100: Code size too large	760
C5200: 'FileName' file not found.	761
C5250: Error in type settings: <Msg>	761
C5300: Limitation: code size '<actualSize>' > '<limitSize>' bytes.	762
C5302: Couldn't open the object file '<FileName>'	762
C5320: Cannot open logfile '<FileName>'	763

C5350: Wrong or invalid encrypted file '<File>' (<MagicValue>).	763
C5351: Wrong encryption file version: '<File>' (<Version>).	764
C5352: Cannot build encryption destination file: '<FileSpec>'	764
C5353: Cannot open encryption source file: '<File>'	765
C5354: Cannot open encryption destination file: '<File>'	765
C5355: Encryption source '<SrcFile>' and destination file '<DstFile>' are the same 766	
C5356: No valid license for encryption support	766
C5650: Too many locations, try to split up function	767
C5651: Local variable <variable> may be not initialized	767
C5660: Removed dead code.	768
C5661: Not all control paths return a value	768
C5662: Generic Inline Assembler Error <string>	769
C5680: HLI Error <Detail>	769
C5681: Unexpected Token	769
C5682: Unknown fixup type	769
C5683: Illegal directive <Ident> ignored.	770
C5684: Instruction or label expected	770
C5685: Comment or end of line expected	770
C5686: Undefined label <Ident>	771
C5687: Label <Ident> defined more than once	771
C5688: Illegal operand.	772
C5689: Constant or object expected	772
C5690: Illegal fixup for constant	772
C5691: Instruction operand mismatch.	773
C5692: Illegal fixup	773
C5693: Cannot generate code for <Error>	773
C5700: Internal Error <ErrorNumber> in '<Module>', please report to <Producer> 774	
C5701: Internal Error #<ErrorNumber> in '<Module>' while compiling file '<File>, procedure '<Function>', please report to <Producer>.	774
C5702: Local variable '<Variable>' declared in function '<Function>' but not referenced	775
C5703: Parameter '<Parameter>' declared in function '<Function>' but not	

referenced	776
C5800: User requested stop	776
C5900: Result is zero	777
C5901: Result is one	777
C5902: Shift count is zero	778
C5903: Zero modulus	778
C5904: Division by one	779
C5905: Multiplication with one	779
C5906: Subtraction with zero	780
C5907: Addition replaced with shift	780
C5908: Constant switch expression	781
C5909: Assignment in condition	781
C5910: Label removed.	782
C5911: Division by zero at runtime	783
C5912: Code in 'if' and 'else' part are the same	783
C5913: Conditions of 'if' and 'else if' are the same	784
C5914: Conditions of 'if' and 'else if' are inverted	785
C5915: Nested 'if' with same conditions	785
C5916: Nested 'if' with inverse conditions	786
C5917: Removed dead assignment	787
C5918: Removed dead goto	787
C5919: Conversion of floating to unsigned integral	788
C5920: Inlining library function <function>	789
C5921: Shift count out of range	789
C6000: Creating Asm Include File <file>	790
C6001: Could not Open Asm Include File because of <reason>	790
C6002: Illegal pragma CREATE_ASM_LISTING because of <reason>	791
Messages of XGATE Back End	791
C22000: XGATE stack pointer is not balanced	792
C22001: XGATE stack frame changed by inline assembler	792
C22002: Wrong type or number of formal parameters in interrupt handler	793

3 ANSI Library Reference	795
Library Files	795
Directory Structure	795
How to Generate Library	795
Common Source Files	796
Startup Files	796
Library Files	797
Special Features	797
Memory Management - malloc, free, calloc, realloc; alloc.c, heap.c	798
Signals - signal.c	798
Multi-byte Characters - mblen, mbtowc, wctomb, mbstowcs, wcstombs; stdlib.c	798
Program Termination - abort, exit, atexit; stdlib.c	798
I/O - printf.c	799
Locales - locale.*	800
ctype	800
String conversions - strtol, strtoul, strtod, stdlib.c	801
Library Structure	801
Error Handling	801
String Handling Functions	802
Memory Block Functions	802
Mathematical Functions	803
Memory Management	805
Searching and Sorting	805
Character Functions	806
System Functions	807
Time Functions	808
Locale Functions	808
Conversion Functions	809
printf and scanf	809
File I/O	809
Types and Macros in the Standard Library	811
errno.h	811

float.h	812
limits.h	813
locale.h	813
math.h	815
setjmp.h	815
signal.h	816
stddef.h	816
stdio.h	817
stdlib.h	818
time.h	818
string.h	819
assert.h	819
stdarg.h	819
ctype.h	820
The Standard Functions	821
abort	822
abs	823
acos, acosf	823
asctime	824
asin, asinf	824
assert	825
atan, atanf	825
atan2, atan2f	826
atexit	827
atof	827
atoi	828
atol	828
bsearch	829
calloc	830
ceil, ceilf	831
clearerr	831
clock	832
cos, cosf	832

Table of Contents

cosh, coshf	833
ctime	833
difftime	834
div	834
exit	835
exp, expf	835
fabs, fabsf	836
fclose	836
feof	837
ferror	837
fflush	838
fgetc	839
fgetpos	839
fgets	840
floor, floorf	841
fmod, fmodf	841
fopen	842
fprintf	843
fputc	844
fputs	844
fread	845
free	845
freopen	846
frexp, frexpf	846
fscanf	847
fseek	847
fsetpos	848
ftell	849
fwrite	849
getc	850
getchar	850
getenv	850
gets	851

gmtime	851
isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit	852
labs	853
ldexp, ldexpf	854
ldiv	854
localeconv	855
localtime	855
log, logf	856
log10, log10f	856
longjmp	857
malloc	857
mblen	858
mbstowcs	858
mbtowc	859
memchr	859
memcmp	860
memcpy, memmove	861
memset	861
mktime	862
modf, modff	862
perror	863
pow, powf	863
printf	864
putc	864
putchar	865
puts	865
qsort	866
raise	867
rand	867
realloc	868
remove	869
rename	869

Table of Contents

rewind	870
scanf	870
setbuf	871
setjmp	871
setlocale	872
setvbuf	873
signal	873
sin, sinf	874
sinh, sinhf	875
sprintf	875
sqrt, sqrtf	879
strrand	879
sscanf	880
strcat	883
strchr	884
strcmp	884
strcoll	885
strcpy	885
strcspn	886
strerror	886
strftime	887
strlen	888
strncat	889
strncmp	889
strncpy	890
strpbrk	890
strrchr	891
strspn	891
strstr	892
strtod	892
strtok	893
strtol	894
strtoul	895

strxfrm	896
system	896
tan, tanf.	897
tanh, tanhf.	897
time	898
tmpfile	898
tmpnam.	899
tolower	899
toupper	900
ungetc	900
va_arg, va_end, va_start	901
vfprintf, vprintf, vsprintf	901
wctomb.	902
wctombcs	903
4 compactC++ Library	905
Introduction	905
Implementation	905
I/O Library	906
Description	906
Headers.	907
Classes	908
Example	908
String Library	909
Description	909
Headers.	909
Classes	909
Example	909
Memory Library	909
Description	909
Headers.	910
Classes	910
Example	910

Table of Contents

Complex Library	910
Description	910
Headers	910
Classes	910
Example	911
Bitset Library	911
Description	911
Headers	911
Classes	911
Example	911
Strstream Library	912
Description	912
Headers	912
Classes	912
Example: #include <strstrea.h>	912
A Porting Tips and FAQs	913
Migration Hints	913
Porting from Cosmic	913
Allocation of Bit Fields	919
Type Sizes, Sign of Character	920
@bool Qualifier	920
@tiny/@far Qualifier for Variables	920
Arrays with Unknown Size	921
Missing Prototype	921
_asm("sequence")	921
Recursive Comments	921
Interrupt Function, @interrupt	922
Defining Interrupt Functions	922
How to Use Variables in EEPROM	925
Linker Parameter File	925
The Application	926
General Optimization Hints	929

Executing an Application from RAM	929
ROM Library Startup File	930
Generate an S Record File	931
Modify the startup code	931
Application PRM file	931
Copying Code from ROM to RAM	932
Invoking the Application Entry Point in the Startup Function.	933
Frequently Asked Questions (FAQs), Trouble Shooting	933
Making Applications	933
Bug Reports	939
Information	939
Bug	939
Critical Bug	940
Electronic Mail (email) or Fax Report Form	940
Technical Support	941
EBNF Notation	941
EBNF Example	942
Terminal Symbols	942
Non-Terminal Symbols	942
Vertical Bar	942
Brackets	943
Parentheses	943
Production End.	943
EBNF-Syntax	943
Extensions	944
Abbreviations, Lexical Conventions	944
Number Formats	944
Precedence and Associativity of Operators for ANSI C	945
List of all Escape Sequences.	947
B Global Configuration File Entries	949
[Options] Section	949
DefaultDir	949

Table of Contents

[XXX_Compiler] Section	950
SaveOnExit	950
SaveAppearance	950
SaveEditor	951
SaveOptions	951
RecentProject0, RecentProject1,	951
TipFilePos	952
ShowTipOfDay	952
TipTimeStamp	953
[Editor] Section	953
Editor_Name	953
Editor_Exe	954
Editor_Opts	954
Example	955

C Local Configuration File Entries 957

[Editor] Section	957
Editor_Name	957
Editor_Exe	958
Editor_Opts	958
Example [Editor] Section	959
[XXX_Compiler] Section	959
RecentCommandLineX	959
CurrentCommandLine	960
StatusbarEnabled	960
ToolbarEnabled	961
WindowPos	961
WindowFont	962
Options	962
EditorType	963
EditorCommandLine	963
EditorDDEClientName	964
EditorDDETTopicName	964

EditorDDEServiceName	964
Example	965
Index	967

Table of Contents

Using the Compiler

Introduction

This document describes the ANSI-C/C++ Compiler. The Compiler consists of a **Front End**, which is language dependent, and a **Back End** that depends on the target processor.

Application Programs

You can find the application programs (build tools) in the \PROG directory where you installed the CodeWarrior software. For example, if you installed the CodeWarrior software in the C:\Metrowerks directory, you can find all application programs in the C:\Metrowerks\PROG directory.

The following list is an overview of the applications used for C/C++ programming.

- `ide.exe` - CodeWarrior IDE
- `cxgate.exe` - Freescale XGATE Compiler
- `axgate.exe` - Freescale XGATE Assembler
- `libmaker.exe` - Librarian Tool to build libraries
- `linker.exe` - Link Tool to build applications (absolute files)
- `decoder.exe` - Decoder Tool to generate assembly listings
- `maker.exe` - Make Tool to rebuild automatically
- `burner.exe` - Batch and interactive Burner (S-Records, ...)
- `hiwave.exe` - Multi-Purpose Simulation/Debugging Environment
- `piper.exe` - Utility to redirect messages to stdout

NOTE

Depending on your license configuration, not all programs listed above may be installed, or there might be additional programs.

Startup Command Line Options

There are some special tool options. These tools are specified at tool startup (while launching the tool). They cannot be specified interactively:

- [-Prod](#) specifies the current project directory or file. For example:

```
linker.exe -Prod=c:\Metrowerks\demo\myproject.pjt
```

There are other options that launch the tool and open its special dialogs. Those dialogs are available in compiler/assembler/burner/maker/linker/decoder/libmaker:

- ShowOptionDialog: This startup option opens the tool option dialog.
- ShowMessageDialog: This startup option opens the tool message dialog.
- ShowConfigurationDialog: This opens the File->Configuration dialog.
- ShowBurnerDialog: This option is for the Burner only and opens the Burner dialog.
- ShowSmartSliderDialog: This option is for the compiler only and opens the smart slider dialog.
- ShowAboutDialog: This option opens the tool about box.

The above options open a modal dialog box where you can specify tool settings. If you press the OK button of the dialog box, the settings are stored in the current project settings file. Example for use:

```
c:\Metrowerks\prog\linker.exe -ShowOptionDialog  
-Prod=c:\demos\myproject.pjt
```

Literature

Refer to the documentation listed below for details about programming languages.

- “American National Standard for Programming Languages – C”, ANSI/ISO 9899–1990 (see also <http://www.iso.org> or ANSI X3.159-1989, X3J11)
- “The C Programming Language”, second edition, Prentice-Hall 1988
- “C: A Reference Manual”, second edition, Prentice-Hall 1987, Harbison and Steele
- “C Traps and Pitfalls”, Andrew Koenig, AT&T Bell Laboratories, Addison-Wesley Publishing Company, Nov. 1988, ISBN 0-201-17928-8
- “Data Structures and C Programs”, Van Wyk, Addison-Wesley 1988
- “How to Write Portable Programs in C”, Horton, Prentice-Hall 1989

- “The UNIX Programming Environment”, Kernighan and Pike, Prentice-Hall 1984
- “The C Puzzle Book”, Feuer, Prentice-Hall 1982
- “C Programming Guidelines”, Thomas Plum, Plum Hall Inc., Second Edition for Standard C, 1989, ISBN 0-911537-07-4
- “DWARF Debugging Information Format”, UNIX International, Programming Languages SIG, Revision 1.1.0 (October 6, 1992), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054
- “DWARF Debugging Information Format”, UNIX International, Programming Languages SIG, Revision 2.0.0 (July 27, 1993), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054
- “System V Application Binary Interface”, UNIX System V, 1992, 1991 UNIX Systems Laboratories, ISBN 0-13-880410-9
- 'Programming Microcontroller in C', Ted Van Sickle, ISBN 1878707140
- 'C Programming for Embedded Systems', Kirk Zurell, ISBN 1929629044
- 'Programming Embedded Systems in C and C ++', Michael Barr, ISBN 1565923545
- 'Embedded C' Michael J. Pont ISBN 020179523X

For programmers not familiar with the C++ language, we recommend reading the additional documentation listed below.

- “Bjarne Stroustrup: The C++ Programming Language, 3rd ed. or special ed.”, 1997 or 2000, Addison-Wesley, ISBN 0201889544 or 0201700735
- “Bjarne Stroustrup: The Design and Evolution of C++”, 1994, ISBN 0-201-54330-3
- “The C++ Standard Library, A Tutorial and Reference”, Nicolai M. Josuttis, <http://www.josuttis.com/libbook/index.html>, Addison-Wesley U.S.A., 1999, ISBN 0-201-37926-0
- “Accelerated C++, Practical Programming by Example”, Andrew Koenig and Barbara E. Moo, <http://www.acceleratedcpp.com/>, Addison-Wesley, 2000, ISBN 0-201-70353-X
- “C++ Templates, The Complete Guide”, David Vandevoorde and Nicolai M. Josuttis, <http://www.josuttis.com/tmplbook/index.html>, Addison-Wesley, 2002, ISBN 0-201-73484-2
- “Standard C++ IOStreams and Locales”, Angelika Langer and Klaus Kreft, <http://www.langer.camelot.de/iostreams.html>, Addison-Wesley, January 2000, ISBN 0-201-18395-1

Highlights

- Powerful User Interface
- Online Help
- Flexible Type Management
- Flexible Message Management
- 32-Bit Application
- Support for Encrypted Files
- High Performance Optimizations
- Conforms to ANSI/ISO 9899-1990

Structure of this Document

- [**User Interface**](#): Description of the Compiler GUI
- [**Compiler Options**](#): Detailed description of the full set of Compiler options
- [**Compiler Predefined Macros**](#): List of all macros predefined by the Compiler
- [**Compiler Pragmas**](#): List of possible pragmas
- [**ANSI-C Front End**](#): Description of the ANSI-C implementation
- [**C++ Front End**](#): C++ language extension compared with ANSI-C with some hints about using C++ in embedded programming (compactC++)
- [**Compiler Back End**](#): Description of code generator and basic type implementation, also hints about hardware oriented programming (optimizations, interrupt functions)
- [**High Level Inline Assembler**](#): Description of the HLI Assembler
- [**Compiler Messages**](#): Description with examples of messages produced by the Compiler
- [**Migration hints**](#): Hints about porting applications from other Compiler vendors to this Compiler
- [**Appendix**](#): FAQs, Trouble shooting, Technical Notes
- Index

Specific Configuration Markers

Some parts of this document are marked with a special indicator box in the upper right corner. Each section that contains an indicator box applies to just the configuration identified by that box. Please check your license configuration.

[Table 1.1](#) lists each type of indicator box.

Table 1.1 Documentation Indicator Boxes

Indicator Box	Description
C++	This section is only available for Compiler with the C++ feature.
XGATE	Only available for Freescale XGATE.
PC	Only available for the IBM-PC operating system.
UNIX	Only available for the UNIX operating system.

CodeWarrior Integration

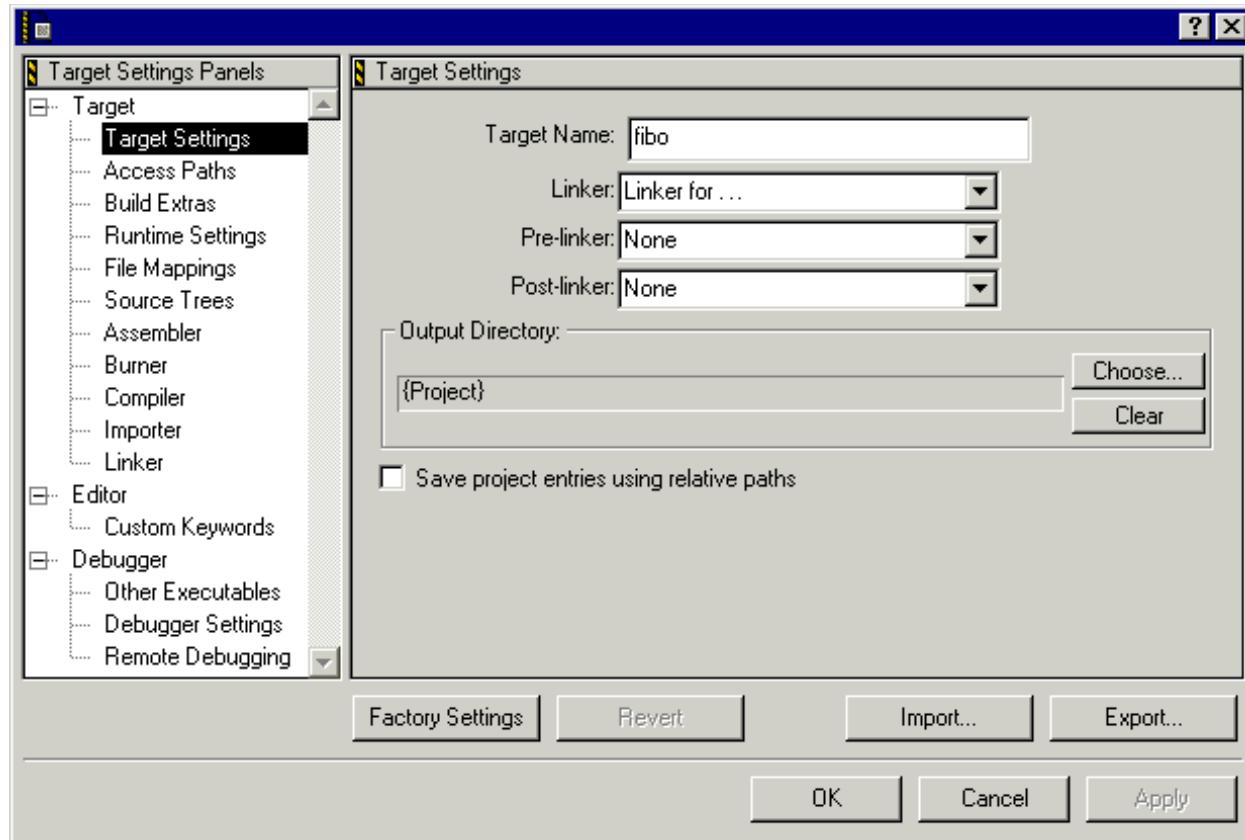
All required plug-ins are installed together with the CodeWarrior IDE. The CodeWarrior IDE is installed in the ‘bin’ directory (usually C:\CodeWarrior\bin). The plug-ins are installed in the ‘bin\plugins’ directory.

Combined/Separated Installations

The installation script enables you to install several CPUs in one single installation path. This saves disk space and enables switching from one processor family to another without leaving the IDE.

NOTE	Additionally, it is possible to have separate installations on one machine. There is only one point to consider: The IDE is using COM files, and for COM the IDE installation path is written into the registry. This registration is done in the installation setup. However, if there is a problem with the COM registration using several installations on one machine, the COM registration is done by starting a small batch file located in the ‘bin’ (usually C:\CodeWarrior\bin) directory. To do this, start the regservers.bat batch file.
-------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

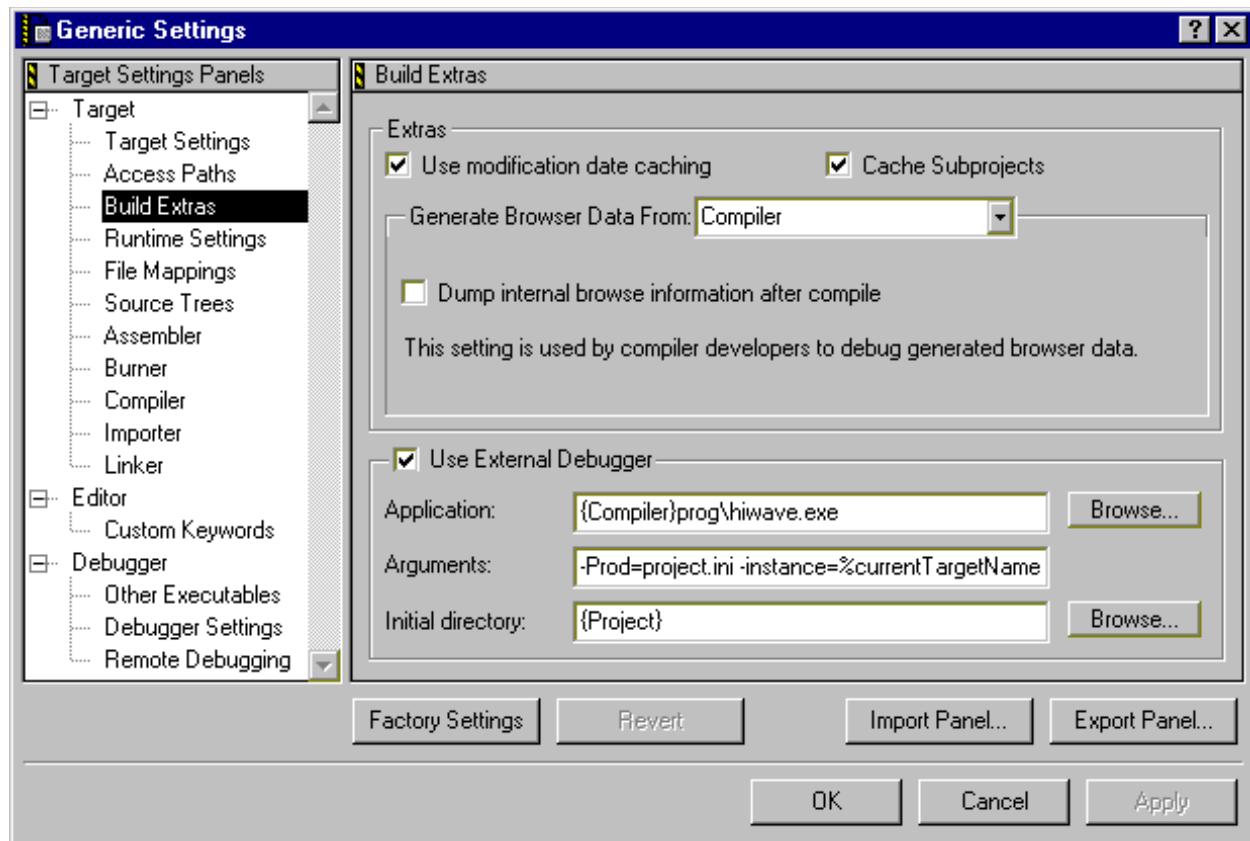
Target Settings



The linker builds an absolute (.abs) file. Before working with a project, set up the linker for the selected CPU in the Target Settings Preference Panel. Depending on the CPU targets installed, you can choose from various linkers available in the linker drop box.

You can also select a libmaker. When a libmaker is set up, the build target is a library (.lib) file.

Build Extras Panel



Use the Build Extras Preference Panel to get the compiler to generate browse information (Activate Browser).

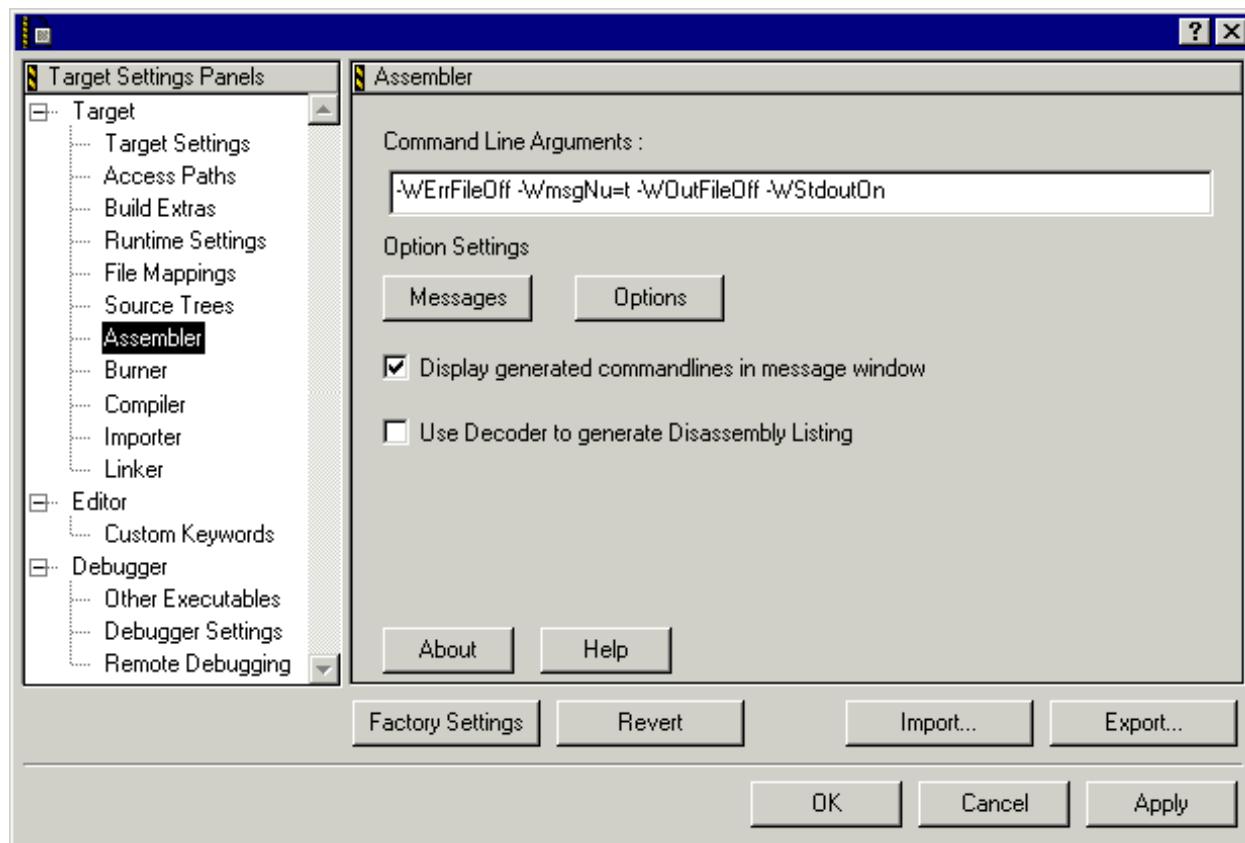
Enable the 'Use third party debugger' checkbox to use the external simulator/debugger. Define the path to the debugger, which is either absolute (for example, 'C:\metrowerks\prog\hiwave.exe'), or installation-relative (for example, '{Compiler}prog\hiwave.exe').

Additional command line arguments passed to the debugger are specified in the Arguments box. Beside the normal arguments (refer to your simulator/debugger documentation), the following '% macros' can also be specified:

```
%sourceFilePath  
%sourceFileDir  
%sourceFileName  
%sourceLineNumber  
%sourceSelection  
%sourceSelUpdate  
%projectFilePath
```

```
%projectFileDir  
%projectFileName  
%projectSelectedFiles  
%targetFilePath  
%targetFileDir  
%targetFileName  
%currentTargetName  
%symFilePath  
%symFileDir  
%symFileName
```

Assembler Preference Panel

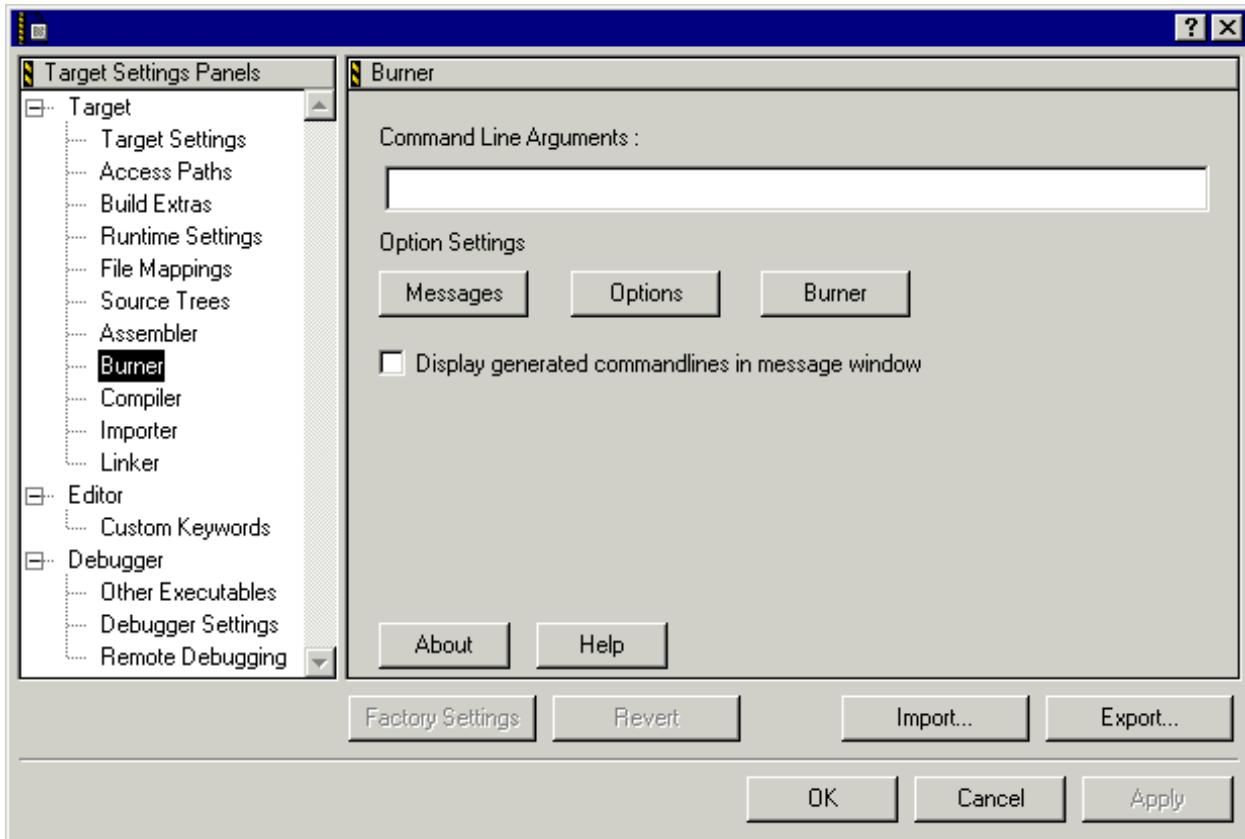


The plug-in preference panel contains the following:

- Command Line Arguments: Command line options are displayed. You can add/delete/modify the options by hand, or by using the Messages and Options buttons below.
- Messages: Button to open the Messages dialog

- Options: Button to open the Options dialog
- Display generated commandlines in message window: The plug-in filters the messages produced, so that only Warning/Information/Error messages are displayed in the ‘Errors & Warnings’ window. With this check box set, the complete command line is passed to the tool.
- Use Decoder to generate Disassembly Listing: The built-in listing file generator is used to produce the disassembly listing. If this check box is set, the external decoder is enabled.
- About: Provides status and version information.
- Help: Opens the help file.

Burner Preference Panel



The Burner Plug-In Function: The *.bbl (batch burner language) files are mapped to the Burner Plug-In in the File Mappings Preference Panel. Whenever a *.bbl file is in the project file, the *.bbl file is processed during the post-link phase using the settings in the Burner Preference-Panel.

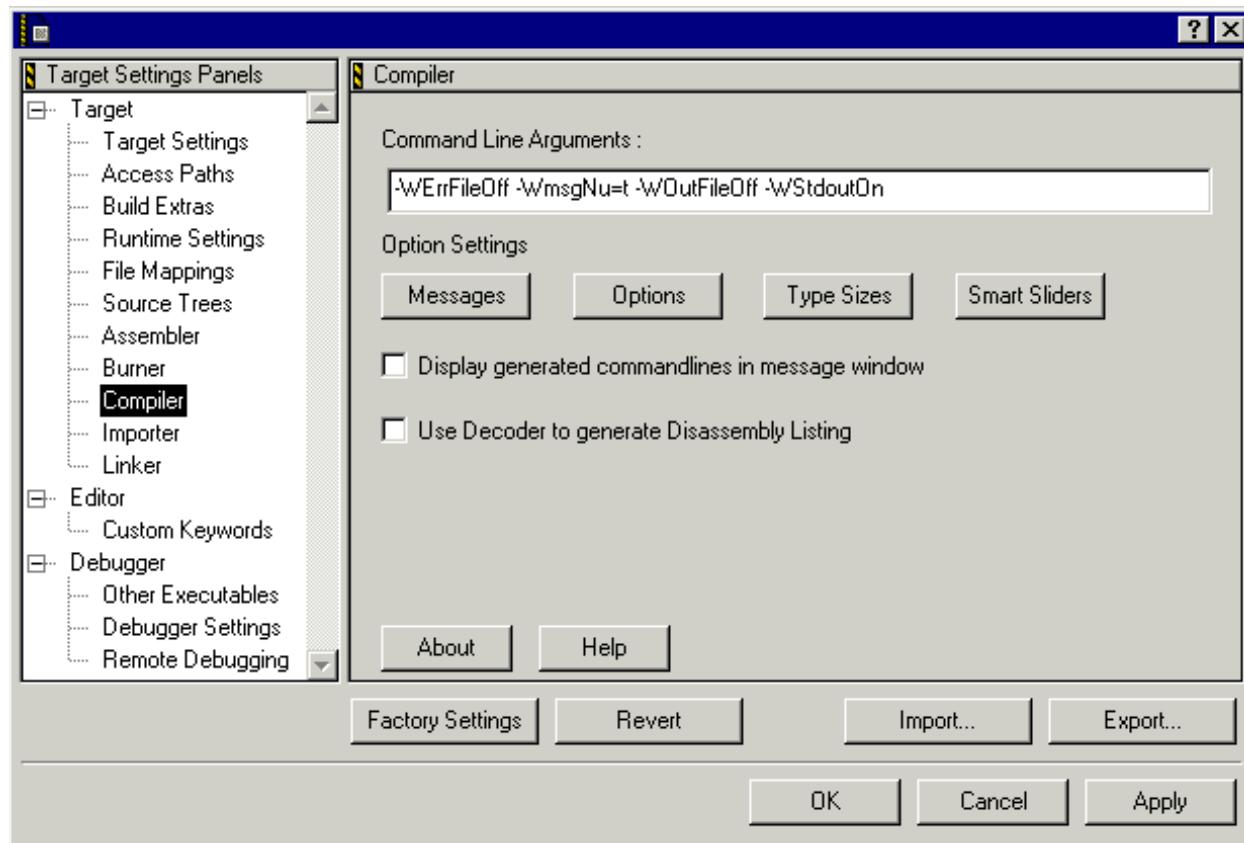
Using the Compiler

Introduction

The plug-in preference panel contains the following:

- Command Line Arguments: The actual command line options are displayed. You can add/delete/modify the options manually, or use the Messages, Options, and Burner buttons listed below.
 - Messages: Opens the Messages dialog
 - Options: Opens the Options dialog
 - Burner: Opens the Burner dialog
- Display generated commandlines in message window: The plug-in filters the messages produced, so that only Warning/Information/Error messages are displayed in the ‘Errors & Warnings’ window. With this check box set, the complete command line is passed to the tool.
- About: Provides status and version information.
- Help: Opens the help file.

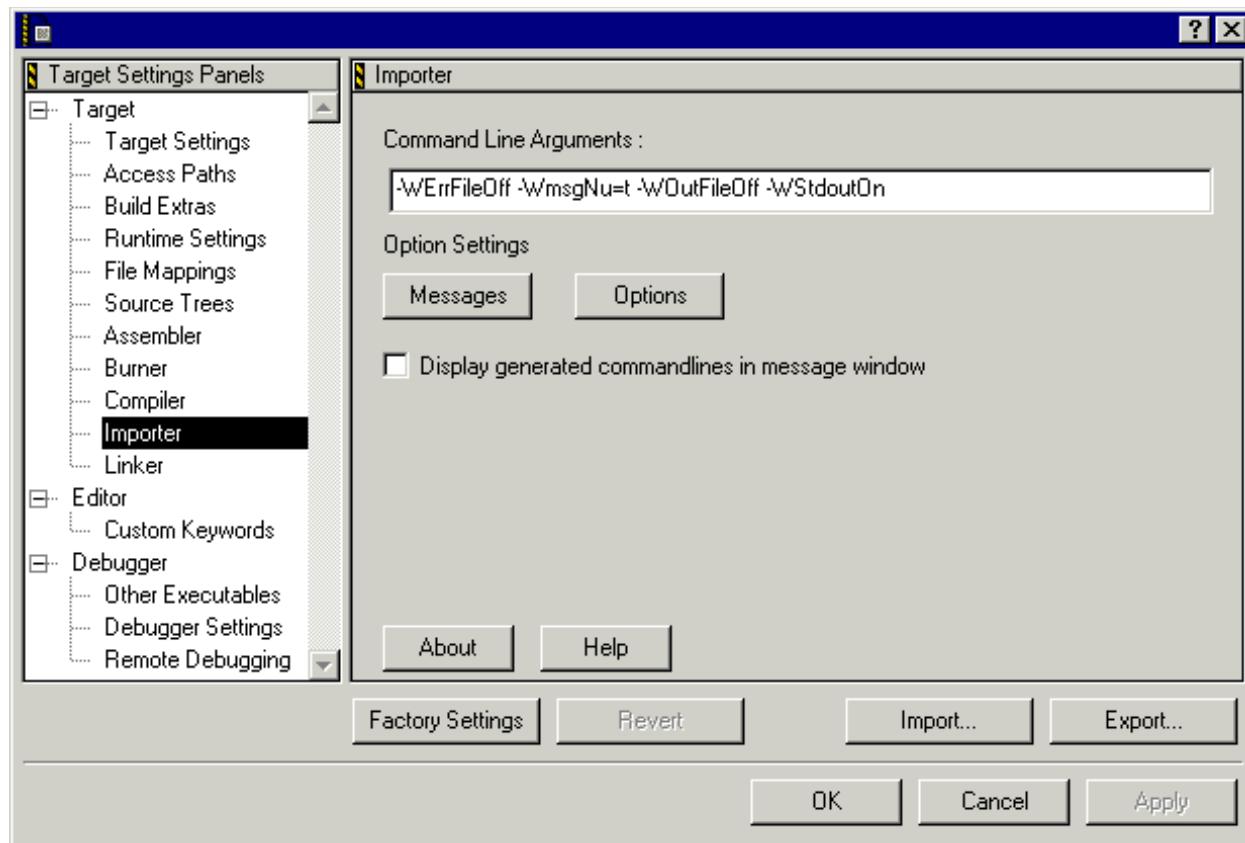
Compiler Preference Panel



The plug-in preference panel contains the following:

- Command Line Arguments: Command line options are displayed. You can add/delete/modify the options manually, or use the Messages, Options, Type Sizes, and Smart Sliders buttons listed below.
 - Messages: Opens the Messages dialog
 - Options: Opens the Options dialog
 - Type Sizes: Opens the Standard Type Size dialog
 - Smart Sliders: Opens the Smart Slider dialog
- Display generated commandlines in message window: The plug-in filters the messages produced, so that only Warning/Information/Error messages are displayed in the ‘Errors & Warnings’ window. With this check box set, the complete command line is passed to the tool.
- Use Decoder to generate Disassembly Listing: Checking this check box enables the external decoder to generate a disassembly listing.
- About: Provides status and version information.
- Help: Opens the help file.

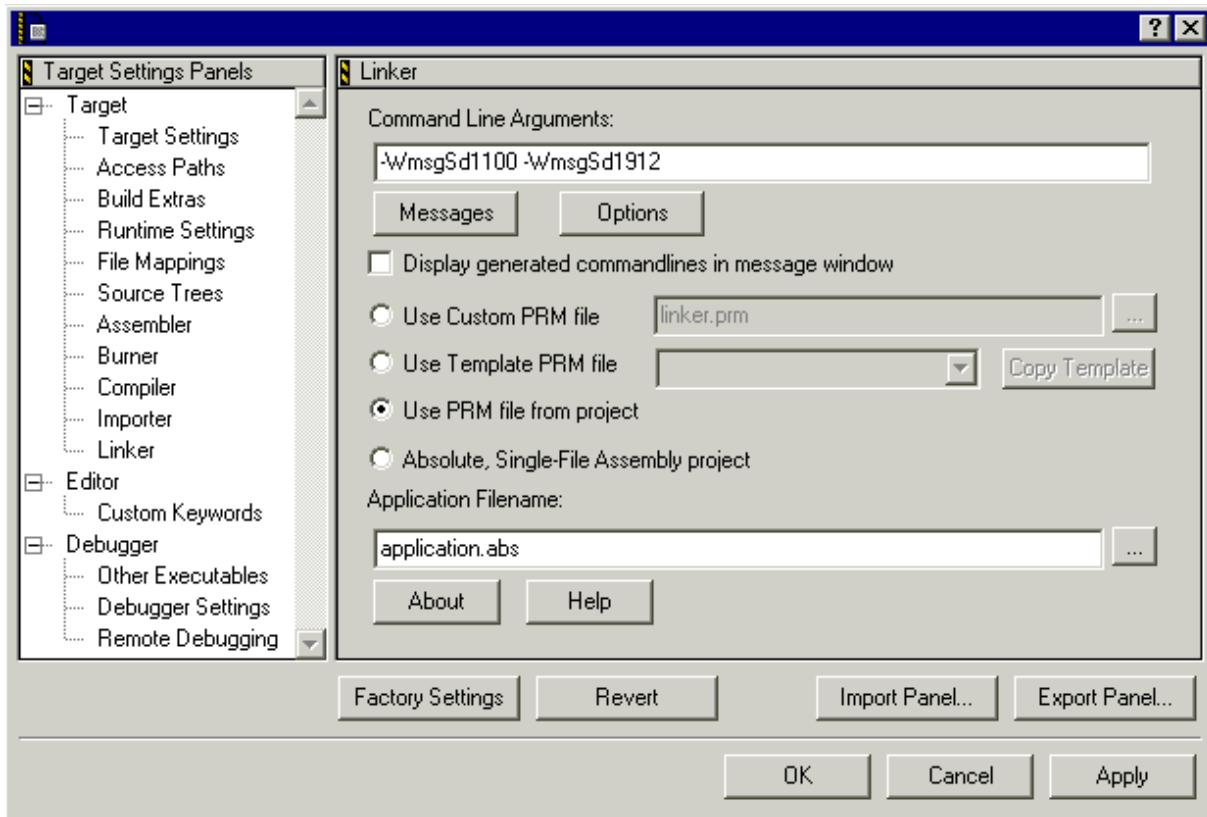
Importer Preference Panel



The plug-in preference panel contains the following controls:

- Command Line Arguments: Command line options are displayed. You can add/delete/modify the options manually, or use the Messages or Options buttons listed below.
 - Messages: Opens the Messages dialog
 - Options: Opens the Options dialog
- Display generated commandlines in message window: The plug-in filters the messages produced so that only Warning/Information/Error messages are displayed in the 'Errors & Warnings' window. With this check box set, the complete command line is passed to the tool.
- About: Provides status and version information.
- Help: Opens the help file.

Linker Preference Panel



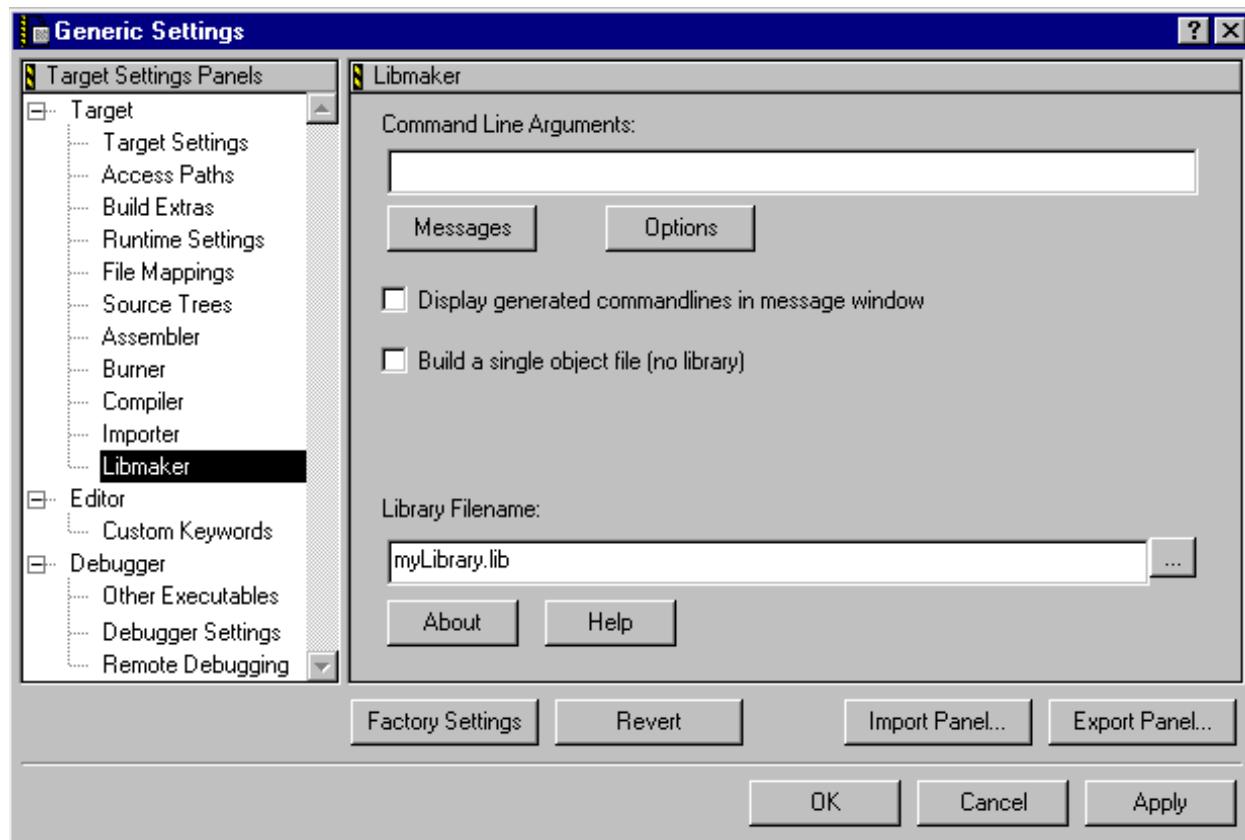
This preference panel displays in the Target Settings Panel if a Linker is selected. The plug-in preference panel contains the following controls:

- Command Line Arguments: Command line options are displayed. You can add/delete/modify the options manually, or use the Messages or Options buttons listed below.
 - Messages: Opens the Messages dialog
 - Options: Opens the Options dialog
- Display generated commandlines in message window: The plug-in filters the messages produced, so that only Warning/Information/Error messages are displayed in the 'Errors & Warnings' window. With this check box set, the complete command line is passed to the tool.
- Use custom PRM file: Specifies a custom linker parameter file in the edit box. Use the browse button (...) to browse for a file.
- Use the template PRM file: With this radio control set, you can select one of the pre-made prm files located in the templates directory (usually

c:\metrowerks\templates\<target>\prm). Additionally using the ‘Copy Template,’ the user can copy a template prm file into the project to maintain a local copy.

- Application File Name: The output file name is specified.
- About: Provides status and version information.
- Help: Button to open the tool help file directly.

Libmaker Preference Panel



This preference panel displays in the Target Settings Panel if a Libmaker is selected. The plug-in preference panel contains the following:

- Command Line Arguments: Command line options are displayed. You can add/delete/modify the options manually, or by using the Messages or Options buttons listed below.
 - Messages: Opens the Messages dialog
 - Options: Opens the Options dialog

- Display generated commandlines in message window: The plug-in filters the messages produced, so that only Warning/Information/Error messages are displayed in the ‘Errors & Warnings’ window. With this check box set, the complete command line is passed to the tool.
- Build a single object file (no library): With this checkbox set, only a single object file is generated. This is useful to generate a startup object file to be linked later, or to create an absolute assembly application.
- Library File Name: The output file name is specified. Use the browse button (directly to the right of the entry field) to locate the directory where the output file is stored.
- About: Provides status and version information.
- Help: Opens the help file.

CodeWarrior Tips and Tricks

- If the simulator debugger can not be launched, check the settings in the Build Extras Preference Panel.
- If the data folder of the project is deleted, then some project-related settings may also have been deleted. One of these settings determines whether the debugger/simulator is enabled. Check the menu Project->Enable/Disable Debugger.
- If a file can not be added to the project, the file extension may be present in the File Mappings Preference Panel.
- If it is suspected that project data is corrupted, export and re-import the project using File->Export Project and File->Import Project.

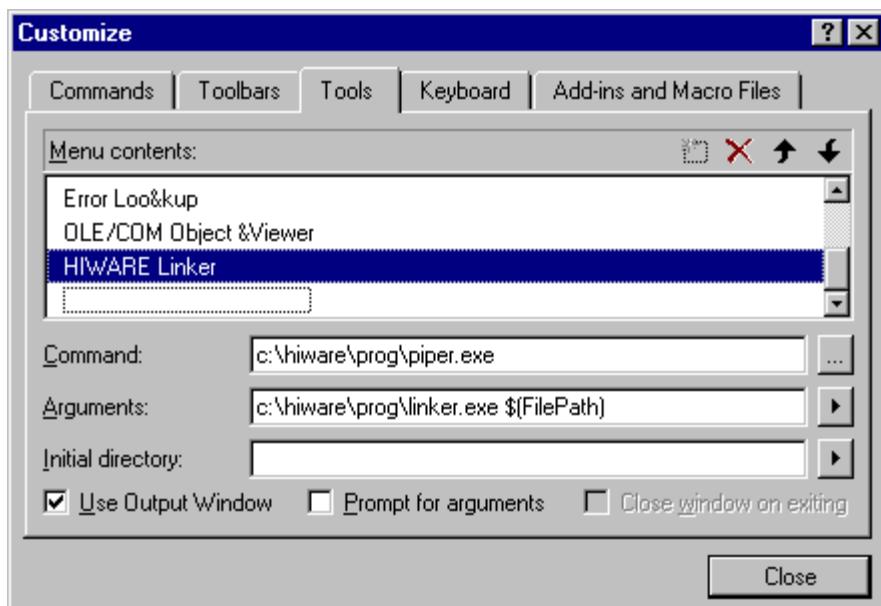
Integration into Microsoft Visual Studio (Visual C++ V5.0 or later)

Use the following procedure to integrate the Tools into the Microsoft Visual Studio (Visual C++).

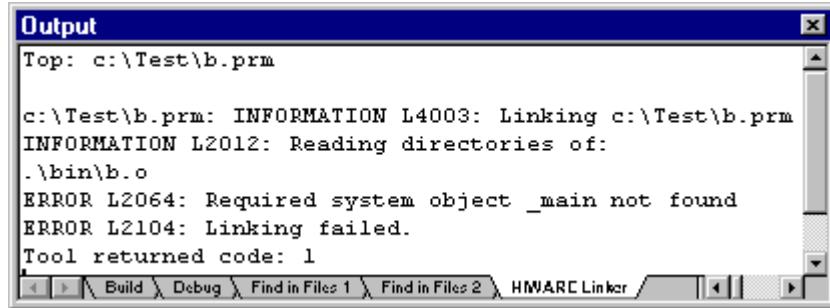
Integration as Additional Tools

1. Start Visual Studio.
2. Select the menu Tools->Customize.
3. Select the ‘Tools’ Tab.

4. Add a new tool using the ‘New’ button, or by double-clicking on the last empty entry in the ‘Menu contents’ list.
5. Type in the name of the tool to display in the menu (for example, ‘Linker’).
6. In the ‘Command’ field, type in the name and path of the piper tool (for example, ‘c:\Metrowerks\prog\piper.exe’).
7. In the ‘Arguments’ field, type in the name of the tool to be started with any command line options (for example, -N) and the \$(FilePath) Visual variable (for example, ‘c:\Metrowerks\prog\linker.exe -N \$(FilePath)’).
8. Check ‘Use Output Window’.
9. Uncheck ‘Prompt for arguments’.
10. Proceed as above for all other tools (for example, compiler, decoder).
11. Close the ‘Customize’ dialog.



This allows the active file to be compiled or linked in the Visual Editor ('\$(FilePath)'). Tool messages are reported in a separate Visual output window. Double click on the output entries to jump to the right message position (message feedback).



Integration with Visual Studio Tool Bar

1. Start Visual Studio.
2. Make sure that all tools are integrated as 'Additional Tools'.
3. Select the menu Tools->Customize.
4. Select the 'Toolbars' Tab.
5. Select 'New' and enter a name (for example, Metrowerks Tools). A new empty toolbar named 'Metrowerks Tools' appears on your screen.
6. Select the 'Commands' Tab.
7. In the 'Category' drop down box, select 'Tools'.
8. On the right side many 'hammer' tool images appear, each with a number. The number corresponds to the entry in the Tool menu. Usually the first user defined tool is tool number 7 (The Linker was set up in 'Additional Tools' above).
9. Drag the selected tool image to the Metrowerks Tools tool bar.
10. All default tool images look the same, making it difficult to know which tool has been launched. It is recommended to associate a name with them.
11. Right-Click on a tool in the Metrowerks Tool bar to open the context menu of the button.
12. Select 'Button Appearance...' in the context menu.
13. Select 'Image and Text'.
14. Enter the tool name to associate with the image in 'Button text:' (for example, 'Linker').
15. Repeat for all tools to appear in the tool bar.
16. Close the 'Customize' Dialog.

This enables the tools to be started from the tool bar.

C++, EC++, compactC++

The Compiler supports the C++ language, if the C++ feature is enabled with a license file.

Some features of the C++ language are not designed for embedded controllers. If they are used, they may produce excess code and require a lot of runtime.

Avoid this situation by providing compactC++ and EC++ images, which are subsets of the C++ language. Each subset is adapted for embedded application programming.

These subsets of the C++ language avoids implicit and explicit overhead of the C++ language (for example, virtual member functions, multiple inheritance). The EC++ is a restricted subset, where the cC++ (compact C++) includes features which are not in the EC++ definition. This makes it more flexible.

Another key aspect of cC++ is its flexible configuration of the language (for example, allowed keywords, code generation behavior, message management). The Compiler is adapted for the special needs for embedded programming.

The Compiler provides the following language settings:

- ANSI-C: The compiler behaves as an ANSI-C compiler. It is possible to force the compiler into a strict ANSI-C compliant mode, or to use language extensions designed for efficient embedded systems programming.
- EC++: The compiler behaves as a C++ compiler. The following features are not allowed in EC++:
 - Mutable specifier
 - Exception handling
 - Runtime type identification
 - Namespace
 - Template
 - Multiple inheritance
 - Virtual inheritance
 - Library support for w_char and long double
- cC++, compactC++: In this mode, the compiler behaves as a full C++ compiler that allows the C++ language to be configured to provide compact code. This enables developers to enable/disable and configure the following C++ features:
 - Multiple inheritance

- Virtual inheritance
 - Templates
 - Trigraph and bigraph
- Compact means:
- No mutable qualifier
 - No exception handling
 - No runtime type identification
 - No namespaces
 - No library support for w_char and long double
- C++: The compiler behaves as a full C++ compiler. However, because the C++ language provides some features not usable for embedded systems programming, such features may be not usable.

Object File Formats

The Compiler supports two different object file formats: ELF/DWARF and the vendor specific HIWARE object file format. The object file format specifies the format of the object files (extension .o), the library files (extension .lib), and the absolute files (extension .abs).

NOTE	Be careful and do not mix object file formats. <i>Both the HIWARE and the ELF/DWARF object files use the same filename extensions.</i>
-------------	----------------------------------------------------------------------------------------------------------------------------------------

HIWARE Object File Format

The HIWARE Object File Format is a vendor-specific object file format defined by HIWARE AG. This object file format is very compact. The object file sizes are smaller than the ELF/DWARF object files. This smaller size enables faster file operations on the object files. The object file format is also easy to support by other tool vendors (for example, emulators from Abatron, Lauterbach or iSYSTEM). The object file format supports ANSI-C and Modula-2. The C++ language is supported with restrictions.

Each other tool vendor must support this object file format explicitly. Note that there is also a lack of extensibility, amount of debug information, and C++ support. For example, using the full flexibility of the Compiler Type Management is not supported in the HIWARE Object File Format.

Using HIWARE object file format may also result in slower source/debug info loading. In the HIWARE object file format, the source position information is provided as position information (offset in file), and not directly in a file/line/column format. The debugger must translate this HIWARE object file source information format into a file/line/column format. This has the tendency to slow down the source file/debugging info loading process.

ELF/DWARF Object File Format

The ELF/DWARF object file format originally comes from the UNIX world. This format is very flexible and is able to support extensions.

Many chip vendors define this object file format as the standard for tool vendors supporting their devices. This standard allows inter-tool operability making it possible to use the compiler from one tool vendor, and the linker from another. The developer has the choice to select the best tool in the tool chain. Additionally, other third parties (for example, emulator vendors) only have to support this object file to support a wide range of tool vendors.

Object file sizes are large compared with the HIWARE object file format.

NOTE ANSI-C, C++ and Modula-2 are supported in this object file format.

Tools

Compiler

The same Compiler executable supports both object file formats. Use the Compiler option [-Fh](#), [-F1](#) or [-F2](#) to switch the object file format.

Note that not all Compiler backends do support both ELF/DWARF and the HIWARE Object File format. Some do only support one of the two.

Decoder

Use the same executable ‘decoder.exe’ for both the HIWARE and the ELF/DWARF object file format.

Linker

Use the same executable ‘linker.exe’ for both the HIWARE and the ELF/DWARF object file formats.

Simulator/Debugger

The Simulator/Debugger supports both object file formats.

Mixing Object File Formats

Mixing HIWARE and ELF object files is not possible. Mixing ELF object files with DWARF 1.1 and DWARF 2.0 debug information is possible. However, the final generated application does not contain any debug data.

Using the Compiler

Introduction

Graphical User Interface

The Graphical User Interface (GUI) tool provides both a simple and a powerful user interface:

- Graphical User Interface
- Command-Line User Interface
- Online Help
- Error Feedback
- Easy integration into other tools (for example, CodeWarrior, CodeWright, MS Visual Studio, WinEdit, ...)

This section describes the user interface and provides useful hints.

Launching a Tool

Start the tools (compiler/linker/assembler/decoder/...) using:

- The Windows Explorer
- An Icon on the desktop
- An Icon in a program group
- Batch/command files
- Other tools (Editor, Visual Studio)

Interactive Mode

If the application (for example, compiler or linker) is started with no input (that means no options and no input files), then the graphical user interface (GUI) is active (interactive mode). This is usually the case if the tool is started using the Explorer or using an Icon.

Batch Mode

If the tool is started with arguments (options and/or input files), then the tool is started in batch mode. Specify the following line associated with an icon on the desktop:

```
C:\Metrowerks\PROG\linker.exe -W2 fibo.prm
```

In batch mode, the tool does not open a window. It is displayed in the taskbar only for the time it processes the input, and terminates afterwards.

Commands are entered to execute as shown below:

```
C:\> C:\Metrowerks\PROG\linker.exe -W2 fibo.prm
```

Message output (stdout) of a tool is redirected using the normal redirection operators (for example, '>' to write the message output to a file), as shown below:

```
C:\> C:\Metrowerks\PROG\linker.exe -W2 fibo.prm >  
myoutput.txt
```

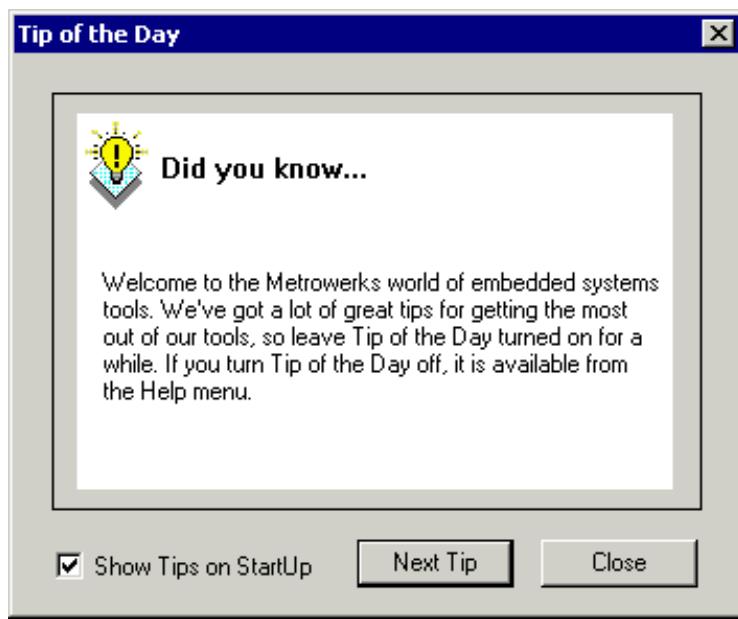
The command line process returns after starting the tool process. It does not wait until the started process has terminated. To start a process and wait for termination (for example, for synchronization), use the 'start' command under Windows NT/95/98, or use the '/wait' options (see windows help 'help start'):

```
C:\> start /wait C:\Metrowerks\PROG\linker.exe -W2 fibo.prm
```

Using 'start /wait' you can write perfect batch files (for example, to process your files).

Tip of the Day

When the application is started, a standard *Tip of the Day* window is opened containing the last news and tips.



The *Next Tip* button displays the next tip about the application.

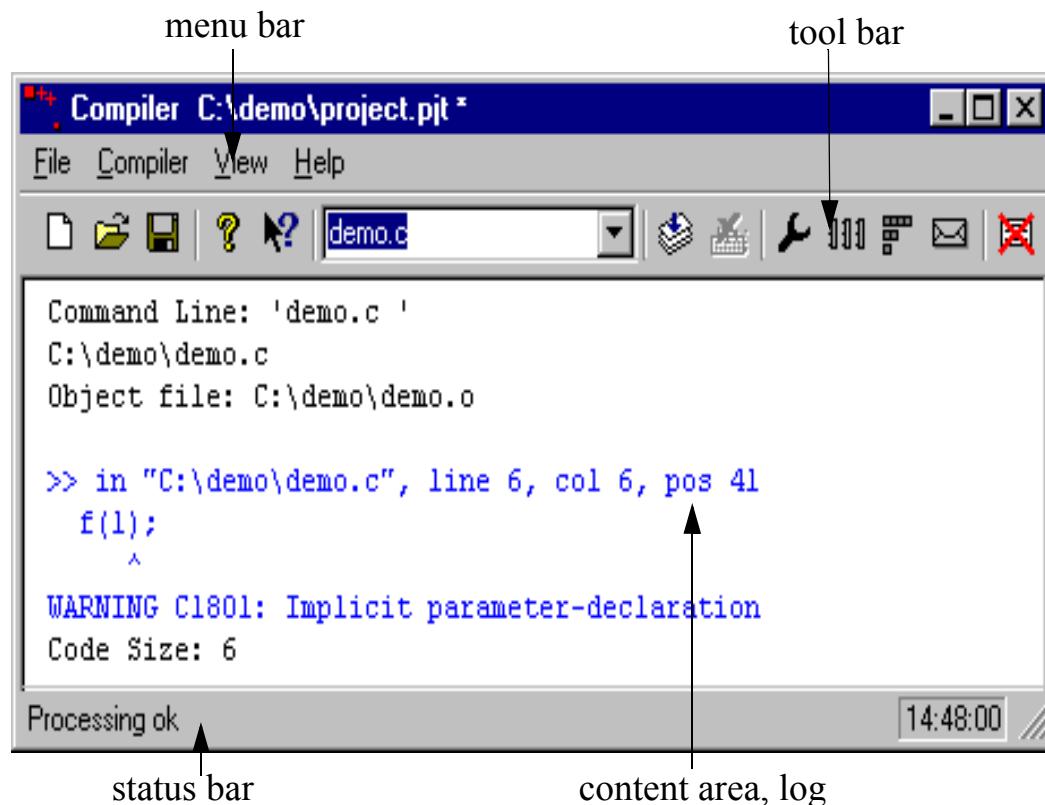
If it is not desired for the *Tip of the Day* window to open automatically when the application is started, uncheck the check box *Show Tips on StartUp*.

NOTE This configuration entry is stored in the local project file.

To enable automatic display from the standard *Tip of the Day* window when the application is started, select the entry *Help | Tip of the Day....* The *Tip of the Day* window will be open. Check the box *Show Tips on StartUp*.

Click *Close* to close the *Tip of the Day* window.

Main Window



This window is only visible on the screen when a file name is not specified while starting the application. The application window provides a window title, a menu bar, a tool bar, a content area, and a status bar.

Window Title

The window title displays the application name and the project name. If there is no project currently loaded, the “Default Configuration” is displayed. An asterisk (*) after the configuration name is present if any value has changed.

NOTE	Changes to options, the editor configuration, and the application appearance can make the “*” appear.
-------------	-------------------------------------------------------------------------------------------------------

Content Area

The content area is used as a text container, where logging information about the process session is displayed. This logging information consists of:

- The name of the file being processed
- The whole names (including full path specifications) of the files processed (main C file and all files included)
- An error, warning and information message list
- The size of the code generated during the process session

When a file is dropped into the application window content area, the corresponding file is either loaded as configuration data, or processed. It is loaded as configuration data if the file has the extension “ini”. If the file does not contain this extension, the file is processed with the current option settings.

All text in the application window content area can contain context information. The context information consists of two items:

- A file name including a position inside of a file
- A message number

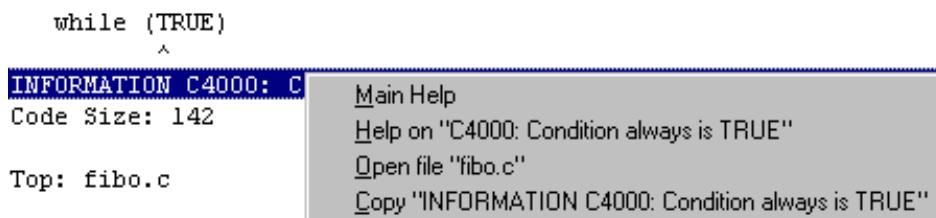
File context information is available for all output where a text file is considered. It is also available for all source and include files, and for messages which do concern a specific file. If a file context is available, double-clicking on the text or message opens this file in an editor, as specified in the editor configuration. The right mouse button can also be used to open a context menu. The context menu contains an “Open ...” entry if a file context is available. If a file can not be opened although a context menu entry is present, refer to the section [Editor Configuration](#).

The message number is available for any message output. There are three ways to open the corresponding entry in the help file.

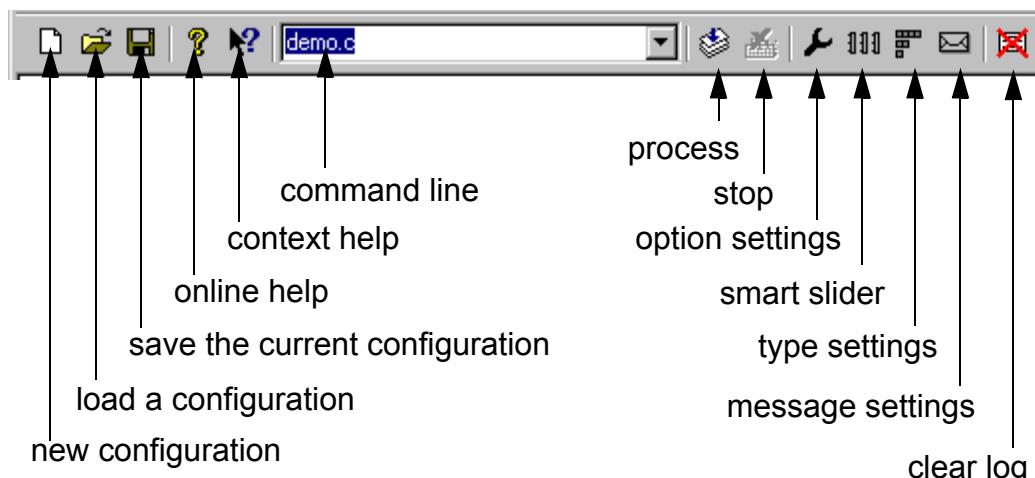
- Select one line of the message and press F1.
If the selected line does not have a message number, the main help is displayed.
- Press Shift-F1 and then click on the message text.
If the point clicked at does not have a message number, the main help is displayed.

- Click with the right mouse at the message text and select “Help on ...”.

This entry is available only if a message number is available.



Tool Bar



The three buttons on the left are linked with the corresponding entries of the *File* menu. The next button opens the *About* dialog. After pressing the context help button (or the shortcut Shift F1), the mouse cursor changes its form and displays a question mark beside the arrow. The help file is called for the next item which is clicked. It is clicked on menus, toolbar buttons and on the window area to get help specific for the topic that is clicked on.

The command line history contains a list of the commands executed. Once a command is selected or entered in history, clicking *Process* starts the execution of the command.

Use the keyboard shortcut F2 key to jump directly to the command line. Additionally there is a context menu associated with the command line:

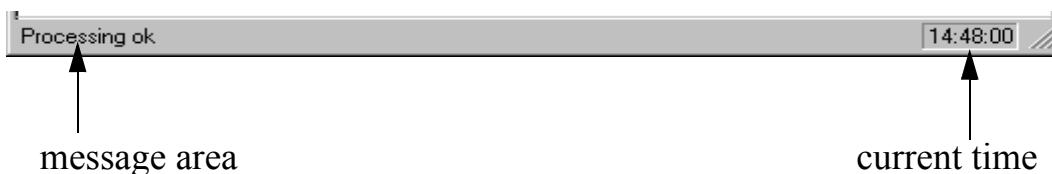


The *Stop* button stops the current process session.

The next four buttons open the option setting, the smart slider, type setting, and the message setting dialog.

The last button clears the content area.

Status Bar



When pointing to a button in the tool bar or a menu entry, the message area displays the function of the button or menu entry being pointed to.

Menu Bar

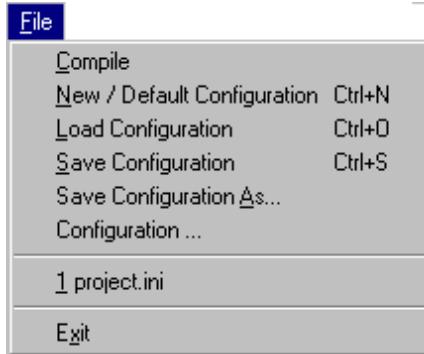


[Table 1.2](#) lists and describes the menus available in the menu bar:

Table 1.2 Menus in the Menu Bar

Menu Entry	Description
File	Contains entries to manage application configuration files.
Compiler	Contains entries to set the application options.
View	Contains entries to customize the application window output.
Help	A standard Windows Help menu.

File Menu



Save or load configuration files from the File Menu. A configuration file contains the following information:

- The application option settings specified in the application dialog boxes
- The Message settings that specify which messages to display and which messages to treat as error messages
- The list of the last command line executed and the current command line being executed
- The window position
- The Tips of the Day settings, including if enabled at startup and which is the current entry

[Configuration files](#) are text files which use the standard extension .ini. A developer can define as many configuration files as required for a project. The developer can also switch between the different configuration files using the *File | Load Configuration* and *File | Save Configuration* menu entries or the corresponding tool bar buttons.

[Table 1.3](#) describes all commands that are available from the File Menu.

Table 1.3 File Menu Commands

Menu entry	Description
Compile	Opens a standard Open File box. The configuration data stored in the selected file is loaded and will be used by a further session.
New / Default Configuration	Resets the application option settings to the default value. The application options which are activated per default are specified in section <i>Command Line Options</i> in this document
Load Configuration	Opens a standard Open File box. The configuration data stored in the selected file is loaded and will be used by a further session.

Table 1.3 File Menu Commands

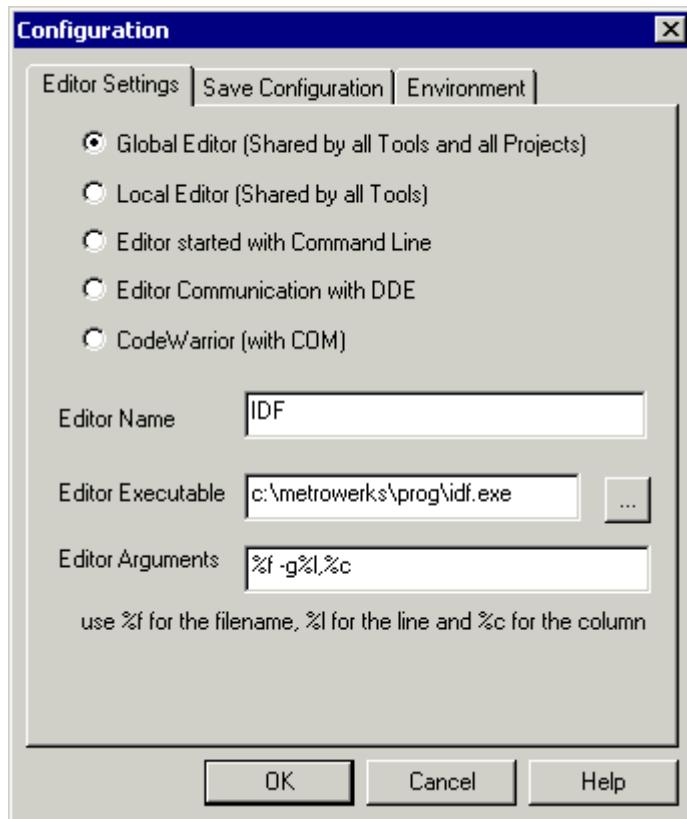
Menu entry	Description
Save Configuration	Saves the current settings.
Save Configuration As...	Opens a standard Save As box. The current settings are saved in a configuration file which has the specified name.
Configuration...	Opens the <i>Configuration</i> dialog box to specify the editor used for error feedback and which parts to save with a configuration.
1. project.ini 2.	Recent project list. This list is accessed to open a recently opened project again.
Exit	Closes the application.

Edit Settings Dialog

The Editor Setting dialog has a main selection entry. Depending on the main type of editor selected, the content below changes.

There main entries are described on the following pages.

- Global Editor

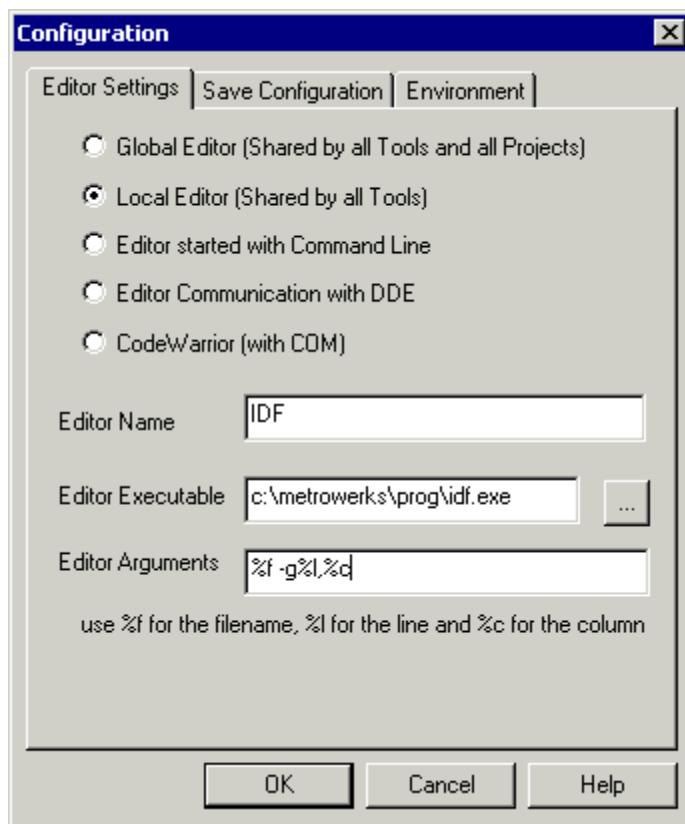


The Global Editor is shared among all tools and projects on one work station. It is stored in the global initialization file "MCUTOOLS.INI" in the "[Editor]" section. Some [Modifiers](#) are specified in the editor command line.

Using the Compiler

Graphical User Interface

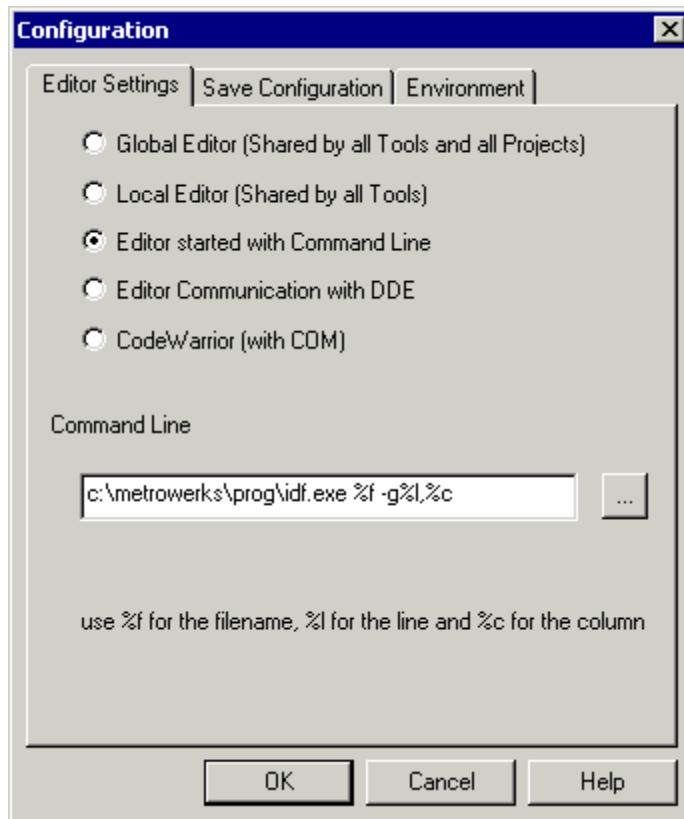
- Local Editor



The local editor is shared among all tools using the same project file. Some [Modifiers](#) are specified in the editor command line.

When an entry of the Global or Local configuration is stored, the behavior of the other tools using the same entry also changes when these tools are restarted.

- Editor started with Command Line



When this editor type is selected, a separate editor is associated with the application for error feedback. The configured editor is not used for error feedback.

Enter the command, that starts the editor.

The format of the editor command depends on the syntax. Some [Modifiers](#) are specified in the editor command line to refer to a file name or a line number (See the Modifiers section below).

The format of the editor command depends on the syntax that is used to start the editor.

Examples: (also look at the notes below)

For Premia CodeWright V6.0 version use (with an adapted path to the cw32.exe file):

C:\Premia\cw32.exe %f -g%l

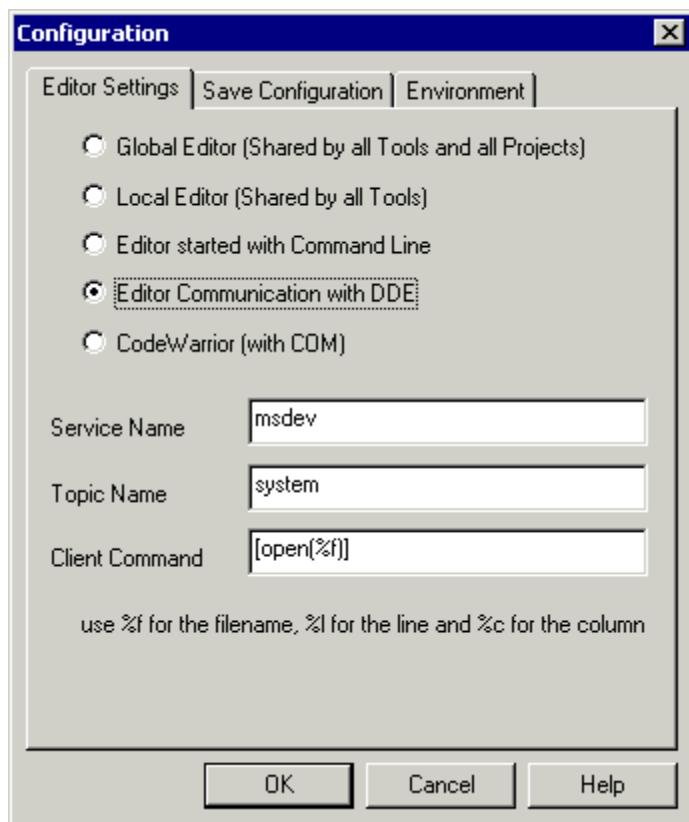
For Winedit 32 bit version use (with an adapted path to the winedit.exe file):

C:\WinEdit32\WinEdit.exe %f /#:%l

Using the Compiler

Graphical User Interface

- Editor started with DDE



Enter the service, topic and client name for the DDE connection to the editor. The entries for Topic and ClientCommand can have modifiers for file name, line number and column number as explained below.

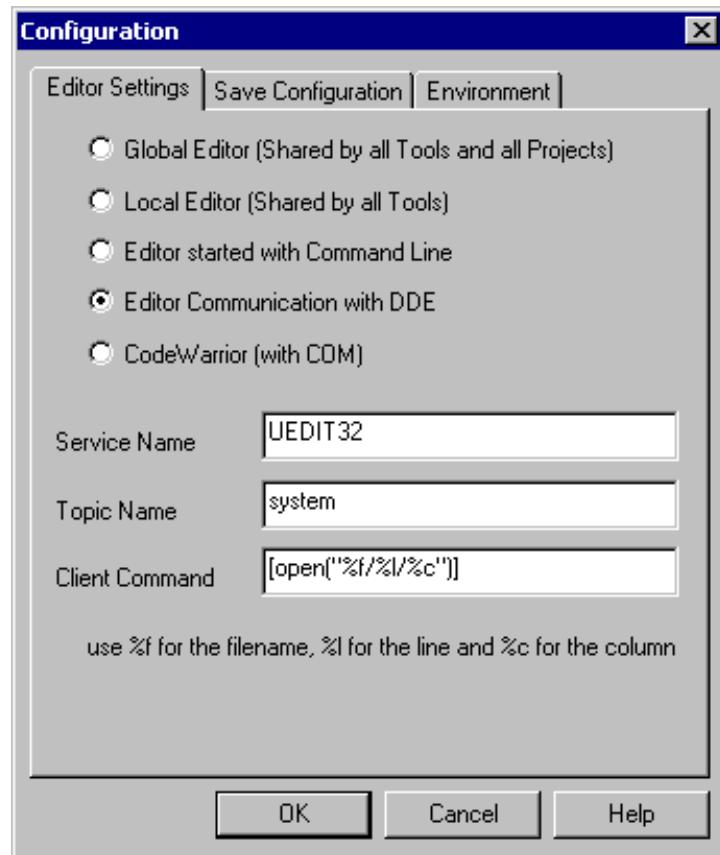
For Microsoft Developer Studio, use the following setting:

```
Service Name      : msdev
Topic Name       : system
ClientCommand    : [ open(%f) ]
```

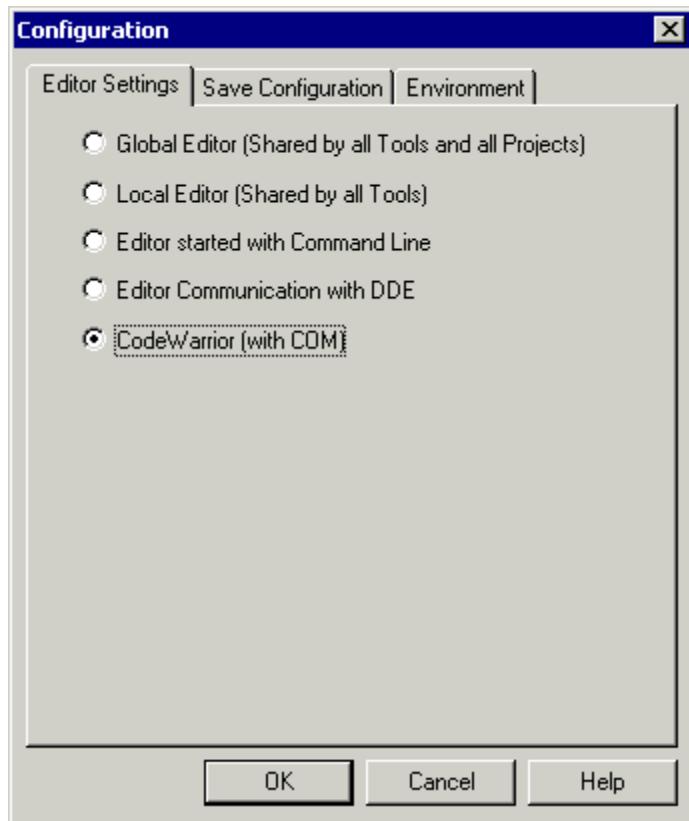
UltraEdit is a powerful shareware editor. It is available from www.idmcomp.com or www.ultraedit.com, email idm@idmcomp.com. The latest version of UltraEdit is also on the CD-ROM in the 'addons' directory. For UltraEdit use the following setting:

```
Service Name      : UEDIT32
Topic Name       : system
ClientCommand    : [ open( "%f/%l/%c" ) ]
```

NOTE The DDE application (Microsoft Developer Studio, UltraEdit) has to be started, otherwise the DDE communication will fail.



- CodeWarrior with COM



If CodeWarrior with COM is enabled, the CodeWarrior IDE (registered as COM server by the installation script) is used as the editor.

Modifiers

The configuration must contain modifiers that instruct the editor which file to open and at which line.

- The %f modifier refers to the name of the file (including path) where the message has been detected.
- The %l modifier refers to the line number where the message has been detected.
- The %c modifier refers to the column number where the message has been detected.

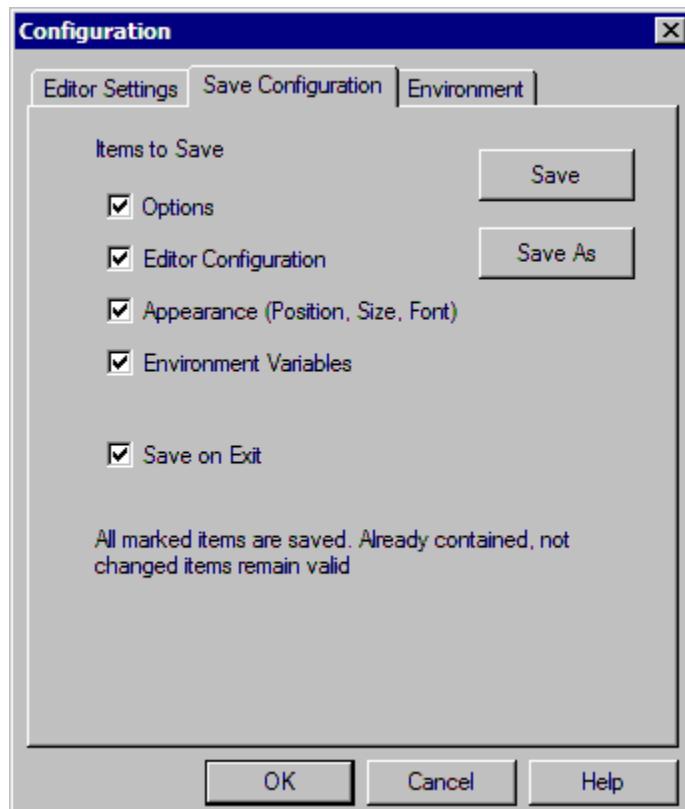
NOTE The %l modifier can only be used with an editor which is started with a line number as parameter. This is not the case for WinEdit version 3.1 or lower or for the Notepad. When working with such an editor, start it with the file name as parameter and then select the menu entry

'Go to' to jump on the line where the message has been detected. *In that case the editor command looks like:*

C:\WINAPPS\WINEDIT\Winedit.EXE %f

Please check the editor manual to define the command line which should be used to start the editor.

Save Configuration Dialog



All save options are located on the second page of the configuration dialog.

Use the In the Save Configuration dialog to configure which parts of your configuration will be stored into a project file.

This Save Configuration dialog offers the following options:

- Options

The current option and message setting is saved when a configuration is written. By disabling this option, the last saved content remains valid.

- Editor Configuration

The current editor setting is saved when a configuration is written. By disabling this option, the last saved content remains valid.

- Appearance

This saves topics consisting of many parts such as the window position (only loaded at startup time) and the command line content and history. These settings are saved when a configuration is written.

- *Environment Variables*

The environment variable changes done in the Environment property sheet are saved.

NOTE	By disabling selective options only some parts of a configuration file are written. For example when the best options are found, the save option mark is removed. Then future save commands will not modify the options any more.
-------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

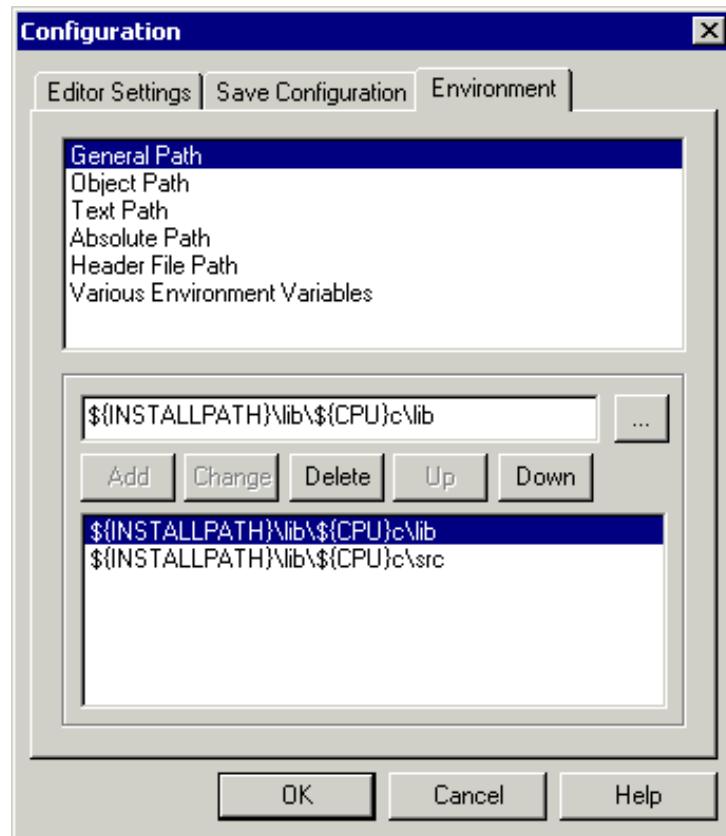
- Save on Exit

The application writes the configuration on exit. No question appears to confirm this operation. If this option is not set, the application will not write the configuration at exit, even if options or another part of the configuration have changed. No confirmation appears in either case when closing the application.

NOTE	Almost all settings are stored in the configuration file only. The only exceptions are: <ul style="list-style-type: none">- The recently used configuration list.- All settings in this dialog.
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

NOTE	The application configurations can (and in fact are intended to) coexist in the same file as the project configuration of CodeWright. When an editor is configured by the shell, the application reads this content out of the project file, if present. The project configuration file of the shell is named project.ini. This file name is also suggested (but not mandatory) to the application.
-------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Environment Configuration Dialog



This Environment Configuration dialog is used to configure the environment. The content of the dialog is read from the actual project file out of the section [Environment Variables]. The following variables are available:

General Path: GENPATH

Object Path: OBJPATH

Text Path: TEXTPATH

Absolute Path: ABSPATH

Header File Path: LIBPATH

Various Environment Variables: other variables not covered by the above list.

The following buttons are available:

Add: Adds a new line/entry

Change: changes a line/entry

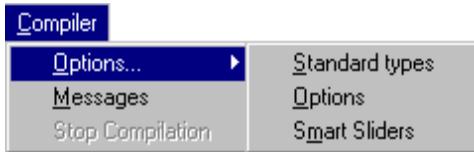
Delete: deletes a line/entry

Up: Moves a line/entry up

Down: Moves a line/entry down

The variables are written to the project file only if the Save Button is pressed (or use File->Save Configuration, or CTRL-S). Additionally, the environment is specified if it is to be written to the project in the Save Configuration dialog.

Compiler Menu

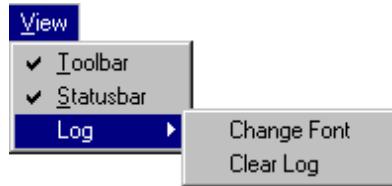


This menu enables the application to be customized. Application options are graphically set as well as defining the optimization level. [Table 1.4](#) defines the Compiler Menu options.

Table 1.4 Compiler Menu Options

Menu entry	Description
Options...	Allows you to customize the application. You can graphically set/reset options. Following entries are available when <i>Options...</i> is selected:
Standard Types	Allows you to specify the size you want to associate with each ANSI C standard type (See <i>Standard Types Dialog Box</i> below).
Advanced	Allows you to define the options which must be activated when processing an input file (See <i>Option Settings Dialog Box</i> below).
Smart Slider	Allows you to define the optimization level you want to reach, when processing the input file (See <i>Compiler Smart Control Dialog Box</i> below).
Messages	Opens a dialog box, where the different error, warning or information messages are mapped to another message class (See <i>Message Setting Dialog Box</i> below).
Stop Compilation	Stops immediately the current processing session.

View Menu



The View Menu enables you to customize the application window. You can define things such as displaying or hiding the status or tool bar. You can also define the font used in the window, or clear the window. [Table 1.5](#) lists the View Menu options.

Table 1.5 View Menu Options

Menu entry	Description
Tool Bar	Switches display from the tool bar in the application window.
Status Bar	Switches display from the status bar in the application window.
Log...	Allows you to customize the output in the application window content area. Following entries are available when <i>Log...</i> is selected:
Change Font	Opens a standard font selection box. The options selected in the font dialog box are applied to the application window content area.
Clear Log	Allows you to clear the application window content area.

Help Menu



The Help Menu enables you to either display or not display the Tip of the Day dialog application startup. Additionally, it provides a standard Windows Help entry and an entry to an About box. [Table 1.6](#) defines the Help Menu options.

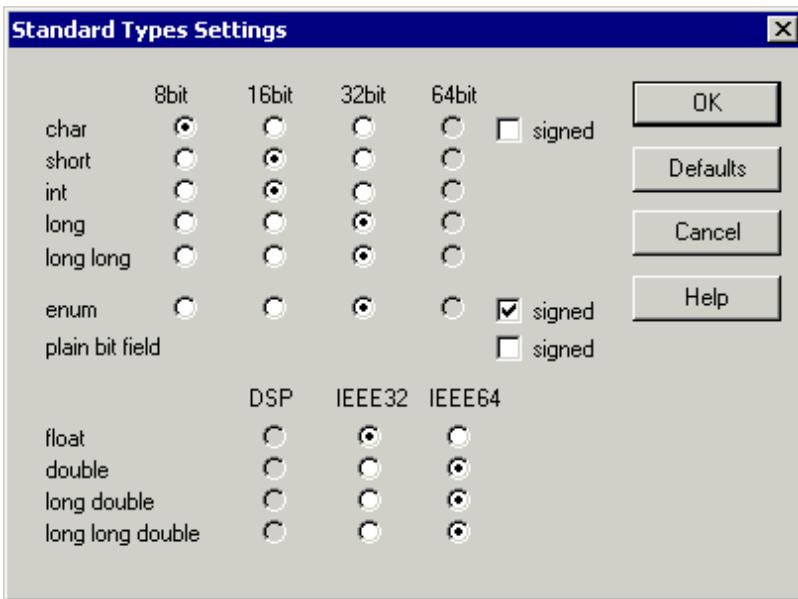
Table 1.6 Help Menu Options

Menu entry	Description
Tip of the Day	Switches on or off the display of a Tip of the Day during startup.

Table 1.6 Help Menu Options

Menu entry	Description
Help Topics	Standard Help topics.
About...	Displays an About box with some version and license information.

Standard Types Dialog Box



The Standard Types Dialog Box enables you to define the size you want to associate to each ANSI C standard type. You can also use the command line option [-T](#) to configure ANSI C standard type sizes.

NOTE Not all formats may be available for a target. Additionally there has to be at least one type for each size. For example, it is illegal to specify all types to a size of 32bits. There is no type for 8bit and 16bit available for the Compiler. Note that if the HIWARE Object File Format is used instead the ELF/DWARF Object File Format, the HIWARE Format does not support a size greater than 1 for the char type.

The following rules apply when you modify the size associated with an ANSI C standard type:

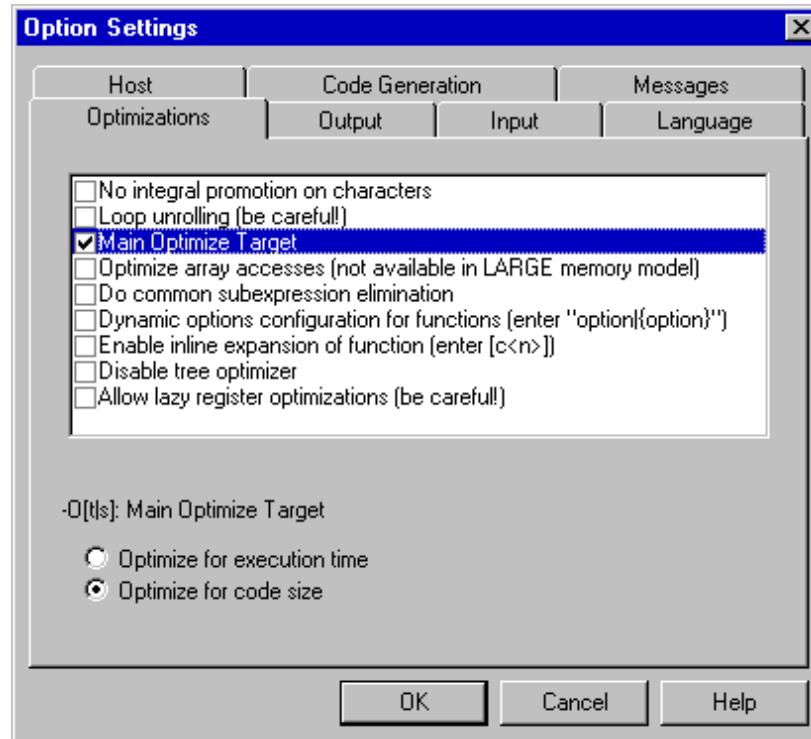
```
sizeof(char)    <= sizeof(short)
sizeof(short)   <= sizeof(int)
sizeof(int)     <= sizeof(long)
sizeof(long)    <= sizeof(long long)
sizeof(float)   <= sizeof(double)
sizeof(double)  <= sizeof(long double)
```

Enumerations must be smaller or equal than ‘int’.

The check box *signed* enables you to specify if type char must be considered as signed or unsigned for your application.

The button *Default* resets the size of the ANSI C standard types to their default values. The ANSI C standard type default values depend on the target processor.

Options Dialog Box



The Options Dialog Box enables you to set or reset application options. The possible command line option is also displayed in the lower display area. The available options

are arranged into different groups. A sheet is available for each of these groups. The content of the list box depends on the selected sheet (not all groups may be available). [Table 1.7](#) lists the Options Dialog Box selections.

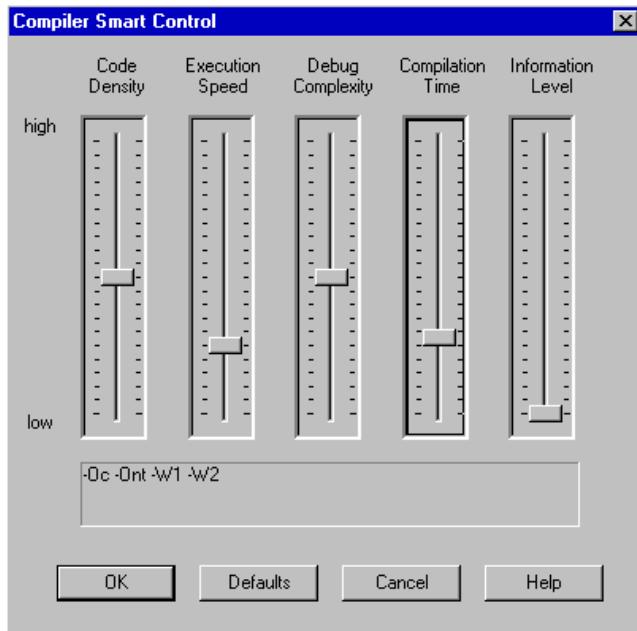
Table 1.7 Options Dialog Box Selections

Group	Description
Optimizations	Lists optimization options.
Output	Lists options related to the output files generation (which kind of file should be generated).
Input	Lists options related to the input file.
Language	Lists options related to the programming language (ANSI C, C++)
Target	Lists options related to the target processor.
Host	Lists options related to the host operating system.
Code Generation	Lists options related to code generation (memory models, float format, ...).
Messages	Lists options controlling the generation of error messages.
Various	Lists options not related to the above options.

An application option is set when its check box is checked. To obtain a more detailed explanation about a specific option, select the option and press the F1 key or the help button. To select an option, click once on the option text. The option text is then displayed color-inverted. When the dialog is opened and no option is selected, pressing the F1 key or the help button shows the help for this dialog.

NOTE	When options requiring additional parameters are selected, you can open an edit box or an additional sub window where the additional parameter is set. For example for the option ‘Write statistic output to file...’, available in the Output sheet.
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Compiler Smart Control Dialog Box



The Compiler Smart Control Dialog Box enables you to define the optimization level you want to reach during compilation of the input file. Five sliders are available to define the optimization level. See [Table 1.8](#).

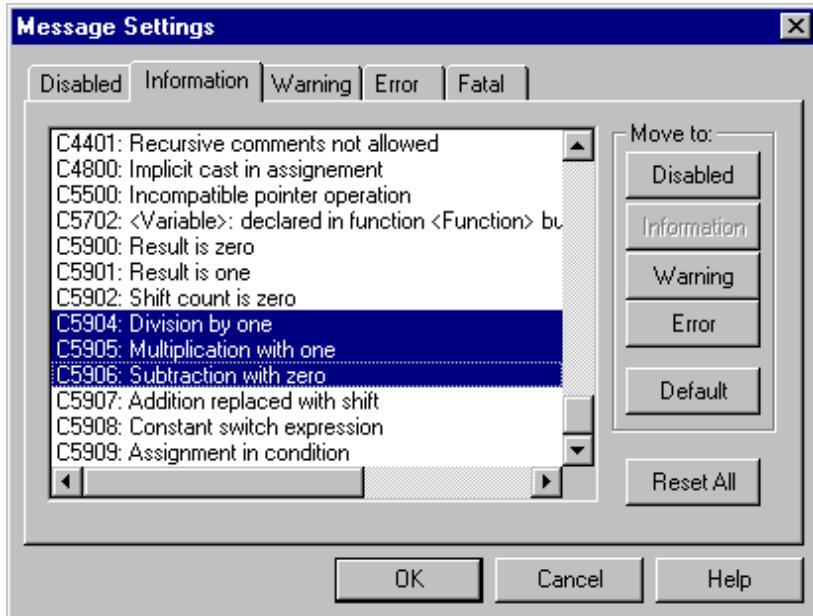
Table 1.8 Compiler Smart Control Dialog Box Controls

Slider	Description
Code Density	Displays the code density level expected. A high value indicates highest code efficiency (smallest code size).
Execution Speed	Displays the execution speed level expected. A high value indicates fastest execution of the code generated.
Debug Complexity	Displays the debug complexity level expected. A high value indicates complex debugging e.g. assembly code corresponds directly to the high level language code.
Compilation Time	Displays the compilation time level expected. A high value indicates long compilation time to produce the object file, e.g. due to high optimization.
Information Level	Displays the level of information messages which are displayed during a Compiler session. A high value indicates a verbose behavior of the Compiler e.g. it will inform with warnings and information messages.

There is a direct link between the first four sliders in this window. When you move one slider, the positions of the other three are updated according to the modification.

The command line is automatically updated with the options set in accordance with the settings of the different sliders.

Message Settings Dialog Box



The Message Settings Dialog Box enables you to map messages to a different message class.

Some buttons in the dialog may be disabled (e.g. if an option cannot be moved to an Information message, the 'Move to: Information' button is disabled). [Table 1.9](#) lists and describes the buttons available in this dialog.

Table 1.9 Message Settings Dialog Box Buttons

Button	Description
Move to: Disabled	The messages selected will be disabled. The message will not occur any longer.
Move to: Information	The messages selected will be changed to information messages.
Move to: Warning	The messages selected will be changed to warning messages.
Move to: Error	The messages selected will be changed to error messages.

Table 1.9 Message Settings Dialog Box Buttons

Button	Description
Move to: Default	The messages selected will be changed to their default message kind.
Reset All	Resets all messages to their default message kind.
Ok	Exits this dialog and accepts the changes made.
Cancel	Exits this dialog without accepting the changes made.
Help	Displays online help about this dialog.

A sheet is available for each error message class. The content of the list box depends on the selected sheet:

Table 1.10 Message Group Definitions

Message group	Description
Disabled	Lists all disabled messages. That means messages displayed in the list box will not be displayed by the application.
Information	Lists all information messages. Information messages inform about action taken by the application.
Warning	Lists all warning messages. When a warning message is generated, processing of the input file continues.
Error	Lists all error messages. When an error message is generated, processing of the input file continues.
Fatal	Lists all fatal error messages. When fatal error message is generated, processing of the input file stops immediately. Fatal messages can not be changed and are only listed to call context help.

Each message has its own character (e.g. ‘C’ for Compiler messages, ‘A’ for Assembler messages, ‘L’ for Linker messages, ‘M’ for Maker messages, ‘LM’ for Libmaker messages) followed by a 4-5 digit number. This number allows an easy search for the message both in the manual or on-line help.

Changing the Class Associated with a Message

You can configure your own mapping of messages in the different classes. For that purpose you can use one of the buttons located on the right hand of the dialog box. Each button refers to a message class. To change the class associated with a message,

you have to select the message in the list box and then click the button associated with the class where you want to move the message.

Example:

To define a warning message as an error message:

1. Click the *Warning* sheet to display the list of all warning messages in the list box.
2. Click on the message you want to change in the list box to select the message.
3. Click *Error* to define this message as an error message.

NOTE Messages cannot be moved to or from the fatal error class.

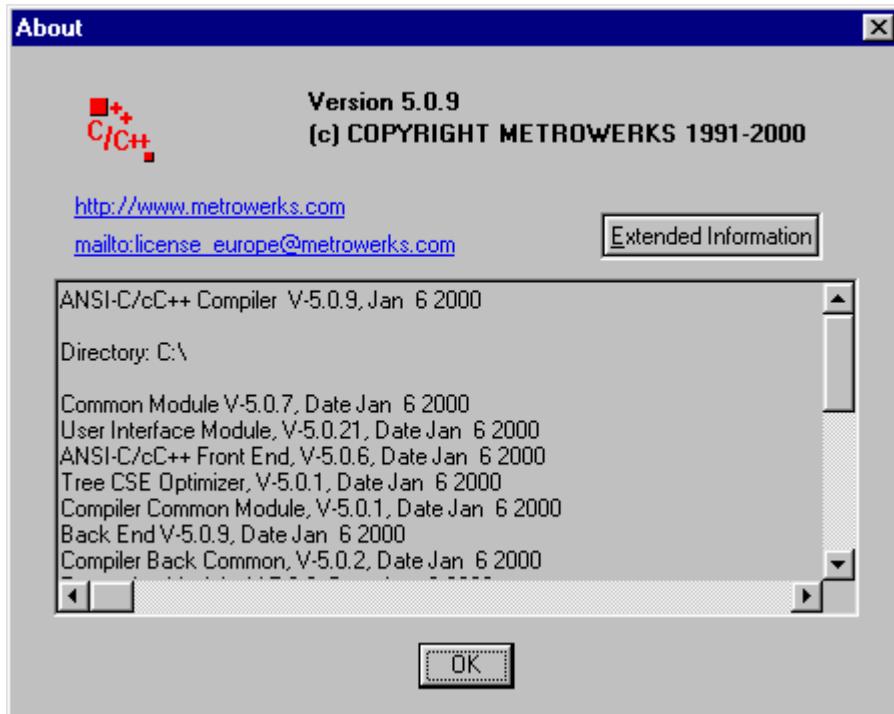
NOTE The ‘move to’ buttons are active only when messages that can be moved are selected. When one message is marked which can not be moved to a specific group, the corresponding ‘move to’ button is disabled (grayed).

If you want to validate the modification you have performed in the error message mapping, close the 'Message settings' dialog box using the 'OK' button. If you close it using the 'Cancel' button, the previous message mapping remains valid.

Retrieving Information About an Error Message

You can access information about each message displayed in the list box. Select the message in the list box and then click *Help* or the F1 key. An information box is opened. The information box contains a more detailed description of the error message, as well as a small example of code that may have generated the error message. If several messages are selected, a help for the first is shown. When no message is selected, pressing the F1 key or the help button shows the help for this dialog.

About Box



The About box is opened by selecting Help->About. The About box contains information regarding your application. The current directory and the versions of subparts of the application are also shown. The main version is displayed separately on top of the dialog.

Use the 'Extended Information' button to get license information about all software components in the same directory as that of the executable file.

Click OK to close this dialog.

NOTE

During processing, the sub-versions of the application parts can not be requested. They are only displayed if the application is inactive.

Specifying the Input File

There are different ways to specify the input file. During the compilation, the options will be set according to the configuration established in the different dialog boxes.

Before starting to compile a file make sure you have associated a working directory with your editor.

Use the Command Line in the Tool Bar to Compile

Compiling a New File

A new file name and additional Compiler options are entered in the command line. The specified file will be compiled as soon as the *Compile* button in the tool bar is selected or the enter key is pressed.

Compiling a File which has already been compiled

The previously executed command is displayed using the arrow on the right side of the command line. A command is selected by clicking on it. It appears in the command line. The specified file is compiled as soon as the *Compile* button in the tool bar is clicked.

Use the Entry File - Compile...

When the menu entry *File | Compile...* is selected, a standard open file box is displayed. Use this to locate the file you want to compile. The selected file is compiled as soon as the standard open file box is closed using the *OK* button.

Use Drag and Drop

A file name is dragged from an external application (for example the File Manager/Explorer) and dropped into the Compiler window. The dropped file is compiled as soon as the mouse button is released in the Compiler window. If a file being dragged has the extension “ini”, it is considered to be a configuration and it is immediately loaded and not compiled. To compile a C file with the extension “ini”, use one of the other methods to compile it.

Message/Error Feedback

There are several ways to check where different errors or warnings have been detected after compilation. The format of the error message is:

```
>> <FileName>, line <line number>, col <column number>, pos  
<absolute position in file>  
<Portion of code generating the problem>  
<message class><message number>: <Message string>
```

Example

```
>> in "C:\DEMO\fibo.c", line 30, col 10, pos 428
    EnableInterrupts
    WHILE (TRUE) {
        (
INFORMATION C4000: Condition always TRUE
```

See also Compiler options [-WmsgFi](#) and [-WmsgFb](#) for different message formats.

Use Information from the Compiler Window

Once a file has been compiled, the Compiler window content area displays the list of all the errors or warnings that were detected.

Use your usual editor to open the source file and correct the errors.

Use a User-Defined Editor

You must first configure the editor you want to use for message/error feedback in the *Configuration* dialog box before you begin the compile process. Once a file has been compiled, double-click on an error message. The selected editor is automatically activated and points to the line containing the error.

Environment

This section describes all environment variables. Some environment variables are also used by other tools (e.g. Linker/Assembler). Consult the respective manual for more information.

Parameters are set in an environment using environment variables. There are three ways to specify your environment:

1. The current project file with the section [Environment Variables]. This file may be specified on Tool startup using the [_Prod](#) option.
2. An optional ‘default.env’ file in the current directory. This file is supported for backwards compatibility. The file name is specified using the [ENVIRONMENT](#) variable. Using the default.env file is not recommended.
3. Setting environment variables on system level (DOS level). This is not recommended.

The syntax is:

```
Parameter = KeyName "=" ParamDef.
```

NOTE	<i>Normally no</i> blanks are allowed in the definition of an environment variable.
-------------	-------------------------------------------------------------------------------------

Example:

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/lib;  
/home/me/my_project
```

Parameters may be defined in several ways:

- Using system environment variables supported by your operating system.
- Putting the definitions into the actual project file in the section named [Environment Variables].
- Putting the definitions in a file called DEFAULT.ENV in the default directory.

NOTE	The maximum length of environment variable entries in the DEFAULT.ENV is 4096 characters.
-------------	-------------------------------------------------------------------------------------------

- Putting the definitions in a file given by the value of the system environment variable ENVIRONMENT.

NOTE The default directory mentioned above is set using the system environment variable [DEFAULTDIR](#).

When looking for an environment variable, all programs first search the system environment, then the DEFAULT.ENV file, and finally the global environment file defined by ENVIRONMENT. If no definition is found, a default value is assumed.

NOTE The environment may also be changed using the [-Env](#) option.

NOTE Ensure there are no spaces at the end of any environment variable declaration.

The Current Directory

The most important environment for all tools is the current directory. The current directory is the base search directory where the tool starts to search for files (e.g. for the DEFAULT.ENV)

The current directory of a tool is determined by the operating system, or by the program which launches another one.

- For the UNIX operating system, the current directory of an launched executable is also the current directory from where the binary file has been started.
- For MS Windows based operating systems, the current directory definition is defined as follows:
 - If the tool is launched using a File Manager/Explorer, the current directory is the location of the launched executable.
 - If the tool is launched using an Icon on the Desktop, the current directory is the one specified and associated with the Icon.
 - If the tool is launched by another launching tool with its own current directory specification (e.g. an editor), the current directory is the one specified by the launching tool (e.g. current directory definition).
- For the tools, the current directory is where the local project file is located. Changing the current project file also changes the current directory if the other project file is in a different directory. Note that browsing for a C file does not change the current directory.

To overwrite this behavior, use the [DEFAULTDIR](#) environment variable.

The current directory is displayed, with other information, using the option “[-V](#)” and in the About box.

Environment Macros

You can use macros in your environment settings.

Example:

```
MyVAR=C:\test
TEXTPATH=$(MyVAR)\txt
OBJPATH=${MyVAR}\obj
```

In the example above, TEXTPATH is expanded to ‘C:\test\txt’ and OBJPATH is expanded to ‘C:\test\obj’. You can use \$() or \${}. However, the referenced variable must be defined.

Special variables are also allowed (special variables are always surrounded by {} and they are case sensitive). Additionally the variable content contains the directory separator ‘\’. The special variables are:

- {Compiler}
That is the path of the executable one directory level up if the executable is ‘c:\metrowerks\prog\linker.exe’, and the variable is ‘c:\metrowerks\’.
- {Project}
Path of the current project file. This is used if the current project file is ‘C:\demo\project.ini’, and the variable contains ‘C:\demo\’.
- {System}
This is the path where your Windows system is installed, e.g. ‘C:\WINNT\’.

Global Initialization File (MCUTOOLS.INI) (PC only)

All tools store some global data into the file MCUTOOLS.INI. The tool first searches for the MCUTOOLS.INI file in the directory of the tool itself (path of the executable). If there is no MCUTOOLS.INI file in this directory, the tool looks for a MCUTOOLS.INI file in the MS Windows installation directory (e.g. C:\WINDOWS).

Example:

```
C:\WINDOWS\MCUTOOLS.INI
```

D:\INSTALL\PROG\MCUTOOLS.INI

If a tool is started in the D:\INSTALL\PROG directory, the project file located in the same directory as the tool is used (D:\INSTALL\PROG\MCUTOOLS.INI).

If the tool is started outside the D:\INSTALL\PROG directory, the project file in the Windows directory is used (C:\WINDOWS\MCUTOOLS.INI).

[“Global Configuration File Entries” on page 949](#) documents the sections and entries you can include in the MCUTOOLS.INI file.

Local Configuration File (usually project.ini)

All the configuration properties are stored in the configuration file. The same configuration file is used by different applications.

The shell uses the configuration file with the name “project.ini” in the current directory only. When the shell uses the same file as the compiler, the editor configuration is written and maintained by the shell and is used by the compiler. Apart from this, the compiler can use any file name for the project file. The configuration file has the same format as the windows .ini files. The compiler stores its own entries with the same section name as those in the global mcutools.ini file. The compiler backend is encoded into the section name, so that a different compiler backend can use the same file without any overlapping. Different versions of the same compiler use the same entries. This plays a role when options, only available in one version, must be stored in the configuration file. In such situations, two files must be maintained for each different compiler version. If no incompatible options are enabled when the file is last saved, the same file may be used for both compiler versions.

The current directory is always the directory where the configuration file is located. If a configuration file in a different directory is loaded, the current directory also changes. When the current directory changes, the entire default.env file is reloaded. When a configuration file is loaded or stored, the options in the environment variable COMPOPTIONS are reloaded and added to the project options. This behavior is noticed when different default.env files exist in different directories, each containing incompatible options in the COMPOPTIONS variable.

When a project is loaded using the first default.env, its COMPOPTIONS are added to the configuration file. If this configuration is stored in a different directory where a default.env exists with incompatible options, the compiler adds options and remarks the inconsistency. You can remove the option from the configuration file with the option settings dialog. You can also remove the option from the

`default.env` with the shell or a text editor, depending which options will be used in the future.

At startup, there are two ways to load a configuration:

- Use the command line option [-Prod](#)
- The file `project.ini` in the current directory

If the [-Prod](#) option is used, the current directory is the directory the project file is in. If the [-Prod](#) option is used with a directory, the `project.ini` file in this directory is loaded.

[“Local Configuration File Entries” on page 957](#) documents the sections and entries you can include in a `project.ini` file.

Paths

Most environment variables contain path lists directing where to look for files. A path list is a list of directory names separated by semicolons. Path names are declared using the following syntax (EBNF Syntax):

```
PathList = DirSpec { ";" DirSpec } .  
DirSpec = [ "*" ] DirectoryName .
```

Example:

```
GENPATH=C:\INSTALL\LIB;D:\PROJECTS\TESTS;/usr/local/lib;/  
home/me/my_project
```

If a directory name is preceded by an asterisk ("*"), the programs recursively search that entire directory tree for a file, not just the given directory itself. The directories are searched in the order they appear in the path list.

Example:

```
LIBPATH=*C:\INSTALL\LIB
```

NOTE	Some DOS environment variables (like GENPATH, LIBPATH, etc.) are used.
-------------	------------------------------------------------------------------------

If you work with CodeWright, you can set the environment using a `<project>.pjt` file in your project directory. This enables you to have different projects in different directories, each with its own environment.

NOTE	When using WinEdit, do <i>not</i> set the system environment variable DEFAULDIR . If you do so, and this variable does not contain the
-------------	--------------------------------------------------------------------------------------------------------------------------------------------------------

project directory given in WinEdit's project configuration, files might not be placed where you expect them to be.

Line Continuation

It is possible to specify an environment variable in an environment file (default.env) over different lines using the line continuation character '\'.

Example:

```
OPTIONS=\  
-W2 \  
-Wpd
```

This is the same as

```
OPTIONS=-W2 -Wpd
```

But this feature may not work well using it together with paths, e.g.

```
GENPATH=.\  
TEXTFILE=.\txt
```

will result in

```
GENPATH=. TEXTFILE=.\txt
```

To avoid such problems, use a semicolon ';' at the end of a path if there is a '\' at the end:

```
GENPATH=.\;  
TEXTFILE=.\txt
```

Environment Variable Details

The remainder of this section describes each of the available environment variables. [Table 1.11](#) lists these the description topics in alphabetical order. Each is divided into several sections.

Table 1.11 Environment Variables—Documentation Topics

Topic	Description
Tools	Lists tools that use this variable.
Synonym	A synonym exists for some environment variables. Those synonyms may be used for older releases of the Compiler and will be removed in the future. A synonym has lower precedence than the environment variable.
Syntax	Specifies the syntax of the option in an EBNF format.
Arguments	Describes and lists optional and required arguments for the variable.
Default	Shows the default setting for the variable or none.
Description	Provides a detailed description of the option and how to use it.
Example	Gives an example of usage, and the effects of the variable where possible. The examples shows an entry in the default.env for a PC.
See also	Names related sections.

COMPOPTIONS: Default Compiler Options

Tools

Compiler

Synonym

HICOMPOPTIONS

Syntax

"COMPOPTIONS=" {<option>}

Arguments

<option>: Compiler command line option

Default

None

Description

If this environment variable is set, the Compiler appends its contents to its command line each time a file is compiled. It is used to globally specify options that should always be set. This frees you from having to specify them at every compilation.

NOTE

It is not recommended to use this environment variable if the Compiler is used (V5.x), because the Compiler adds the options specified in the COMPOPTIONS variable to the options stored in the project.ini.

Example

COMPOPTIONS=-W2 -Wpd

See also

[Compiler options](#)

COPYRIGHT: Copyright Entry in Object File

Tools

Compiler, Assembler, Linker, Librarian

Synonym

None

Syntax

"COPYRIGHT=" <copyright>

Arguments

<copyright>: copyright entry

Default

None

Description

Each object file contains an entry for a copyright string. This information is retrieved from the object files using the decoder.

Example

COPYRIGHT=Copyright by Metrowerks

See also

[Environment variable USERNAME](#)

[Environment variable INCLUDETIME](#)

PC

DEFAULTDIR: Default Current Directory

Tools

Compiler, Assembler, Linker, Decoder, Debugger, Librarian, Maker, Burner

Synonym

None

Syntax

"DEFAULTDIR=" <directory>

Arguments

<directory>: Directory to be the default current directory

Default

None

Description

Specifies the default directory for all tools. All the tools indicated above will take the specified directory as their current directory instead of the one defined by the operating system or launching tool (e.g. editor).

NOTE	This is an environment variable on a system level (global environment variable). It can not be specified in a default environment file (DEFAULT.ENV).
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------

Example

DEFAULTDIR=C:\INSTALL\PROJECT

See also

[Section ‘The Current Directory’](#)

[Section MCUTOOLS.INI File](#)

ENVIRONMENT: Environment File Specification

PC

Tools

Compiler, Linker, Decoder, Debugger, Librarian, Maker, Burner

Synonym

HIENVIRONMENT

Syntax

"ENVIRONMENT=" <file>

Arguments

<file>; file name with path specification, without spaces

Default

None

Description

This variable is specified on a system level. The application looks in the [current directory](#) for a environment file named default.env. Using ENVIRONMENT (e.g. set in the autoexec.bat (DOS) or .cshrc (UNIX)), a different file name may be specified.

NOTE

This is an environment variable on a system level (global environment variable). It can not be specified in a default environment file (DEFAULT.ENV).

Example

ENVIRONMENT=\Metrowerks\prog\global.env

See also

None

ERRORFILE: Error File Name Specification

Tools

Compiler, Assembler, Linker, Burner

Synonym

None

Syntax

"ERRORFILE=" <file name>

Arguments

<file name>: File name with possible format specifiers

Description

The ERRORFILE environment variable specifies the name for the error file.

Possible format specifiers are:

'%n': Substitute with the file name, without the path.

'%p': Substitute with the path of the source file.

'%f': Substitute with the full file name, i.e. with the path and name (the same as '%p%n').

A notification box is shown in the event of an illegal error file name.

Example

ERRORFILE=MyErrors.err

Lists all errors into the file MyErrors.err in the current directory.

ERRORFILE=\tmp\errors

Lists all errors into the file errors in the directory \tmp.

ERRORFILE=%f.err

Lists all errors into a file with the same name as the source file, but with extension .err, into the same directory as the source file. If you compile a file such as \sources\test.c, an error list file, \sources\test.err, is generated.

```
ERRORFILE=\dir1\%n.err
```

For a source file such as test.c, an error list file \dir1\test.err is generated.

```
ERRORFILE=%p\errors.txt
```

For a source file such as \dir1\dir2\test.c, an error list file \dir1\dir2\errors.txt is generated.

If the environment variable ERRORFILE is not set, the errors are written to the file EDOUT in the current directory.

Example

Another example shows the usage of this variable to support correct error feedback with the WinEdit Editor. The editor looks for an error file named EDOUT, as shown:

```
Installation directory: E:\INSTALL\PROG
Project sources: D:\MEPHISTO
Common Sources for projects: E:\CLIB
```

```
Entry in default.env (D:\MEPHISTO\DEFAULT.ENV):
ERRORFILE=E:\INSTALL\PROG\EDOUT
```

```
Entry in WINEDIT.INI (in Windows directory):
OUTPUT=E:\INSTALL\PROG\EDOUT
```

NOTE

Be careful to set this variable if the WinEdit Editor is used, otherwise the editor cannot find the EDOUT file.

See also

None

GENPATH: #include “File” Path

Tools

Compiler, Linker, Decoder, Debugger, Burner

Synonym

HIPATH

Syntax

"GENPATH=" {<path>}

Arguments

<path>: Paths separated by semicolons, without spaces

Default

Current directory

Description

If a header file is included with double quotes, the Compiler searches first in the current directory, then in the directories listed by GENPATH, and finally in the directories listed by [LIBRARYPATH](#).

NOTE	If a directory specification in this environment variable starts with an asterisk (“*”), the whole directory tree is searched recursively depth first, i.e. all subdirectories and <i>their</i> subdirectories and so on are searched. Search order of the subdirectories is indeterminate within one level in the tree.
-------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Example

GENPATH=\sources\include;..\..\headers;\usr\local\lib

See also

[Environment variable LIBRARYPATH](#)

INCLUDETIME: Creation Time in Object File

Tools

Compiler, Assembler, Linker, Librarian

Synonym

None

Syntax

"INCLUDETIME=" ("ON" | "OFF")

Arguments

"ON": Include time information into object file

"OFF": Do not include time information into object file

Default

"ON"

Description

Each object file contains a time stamp indicating the creation time and date as strings. Whenever a new file is created by one of the tools, the new file gets a new time stamp entry.

This behavior may be undesired if (for Software Quality Assurance reasons) a binary file compare has to be performed. Even if the information in two object files is the same, the files do not match exactly as the time stamps are not identical. To avoid such problems, set this variable to OFF. In this case, the time stamp strings in the object file for date and time are “none” in the object file.

The time stamp is retrieved from the object files using the decoder.

Example

INCLUDETIME=OFF

See also

[Environment variable COPYRIGHT](#)

[Environment variable USERNAME](#)

LIBRARYPATH: ‘include <File>’ Path

Tools

Compiler, ELF tools (Burner, Linker, Decoder)

Synonym

LIBPATH

Syntax

"LIBRARYPATH=" {<path>}

Arguments

<path>: Paths separated by semicolons, without spaces

Default

Current directory

Description

If a header file is included with double quotes, the Compiler searches first in the current directory, then in the directories given by [GENPATH](#) and finally in the directories given by LIBRARYPATH.

NOTE	If a directory specification in this environment variable starts with an asterisk (“*”), the whole directory tree is searched recursively depth first, i.e. all subdirectories and <i>their</i> subdirectories and so on are searched. Search order of the subdirectories is indeterminate within one level in the tree.
-------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Example

LIBRARYPATH=\sources\include;..\..\headers;\usr\local\lib

See also

[Environment variable GENPATH](#)

[Environment variable USELIBPATH](#)

[Section “Input Files”](#)

OBJPATH: Object File Path

Tools

Compiler, Linker, Decoder, Debugger, Burner

Synonym

None

Syntax

"OBJPATH=" <path>

Default

Current directory

Arguments

<path>: Path without spaces

Description

If the Compiler generates an object file, the object file is placed into the directory specified by OBJPATH. If this environment variable is empty or does not exist, the object file is stored into the path where the source has been found.

If the Compiler tries to generate an object file specified in the path specified by this environment variable but fails (e.g. because the file is locked), the Compiler will issue an error message.

If a tool (e.g. the Linker) looks for an object file, it first checks for an object file specified by this environment variable, then in [GENPATH](#), and finally in HIPATH.

Example

OBJPATH=\sources\obj

See also

[Section “Output Files”](#)

TEXTPATH: Text File Path

Tools

Compiler, Linker, Decoder

Synonym

None

Syntax

"TEXTPATH=" <path>

Arguments

<path>: Path without spaces

Default

Current directory

Description

If the Compiler generates a textual file, the file is placed into the directory specified by TEXTPATH. If this environment variable is empty or does not exist, the text file is stored into the current directory.

Example

TEXTPATH=\sources\txt

See also

[Section “Output Files”](#)

[Compiler option -Ll](#)

[Compiler option -Lm](#)

[Compiler option -Lo](#)

TMP: Temporary Directory

PC

Tools

Compiler, Assembler, Linker, Debugger, Librarian

Synonym

None

Syntax

"TMP=" <directory>

Arguments

<directory>: Directory to be used for temporary files

Default

None

Description

If a temporary file must be created, the ANSI function, `tmpnam()`, is used. This library function stores the temporary files created in the directory specified by this environment variable. If the variable is empty or does not exist, the current directory is used. Check this variable if you get the error message “Cannot create temporary file”.

NOTE	This is an environment variable on a system level (global environment variable). It can not be specified in a default environment file (DEFAULT.ENV).
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------

Example

`TMP=C:\TEMP`

See also

[Section ‘The Current Directory’](#)

USELIBPATH: Using LIBPATH Environment Variable

Tools

Compiler, Linker, Debugger

Synonym

None

Syntax

"USELIBPATH=" ("OFF" | "ON" | "NO" | "YES").

Arguments

"ON", "YES": The environment variable [LIBRARYPATH](#) is used by the Compiler to look for system header files <*.h>.

"NO", "OFF": The environment variable [LIBRARYPATH](#) is not used by the Compiler.

Default

ON

Description

This environment variable allows a flexible usage of the [LIBRARYPATH](#) environment variable as the [LIBRARYPATH](#) variable might be used by other software (e.g. version management PVCS).

Example

USELIBPATH=ON

See also

[Environment variable LIBRARYPATH](#)

USERNAME: User Name in Object File

Tools

Compiler, Assembler, Linker, Librarian

Synonym

None

Syntax

"USERNAME=" <user>

Arguments

<user>: Name of user

Default

None

Description

Each object file contains an entry identifying the user who created the object file. This information is retrievable from the object files using the decoder.

Example

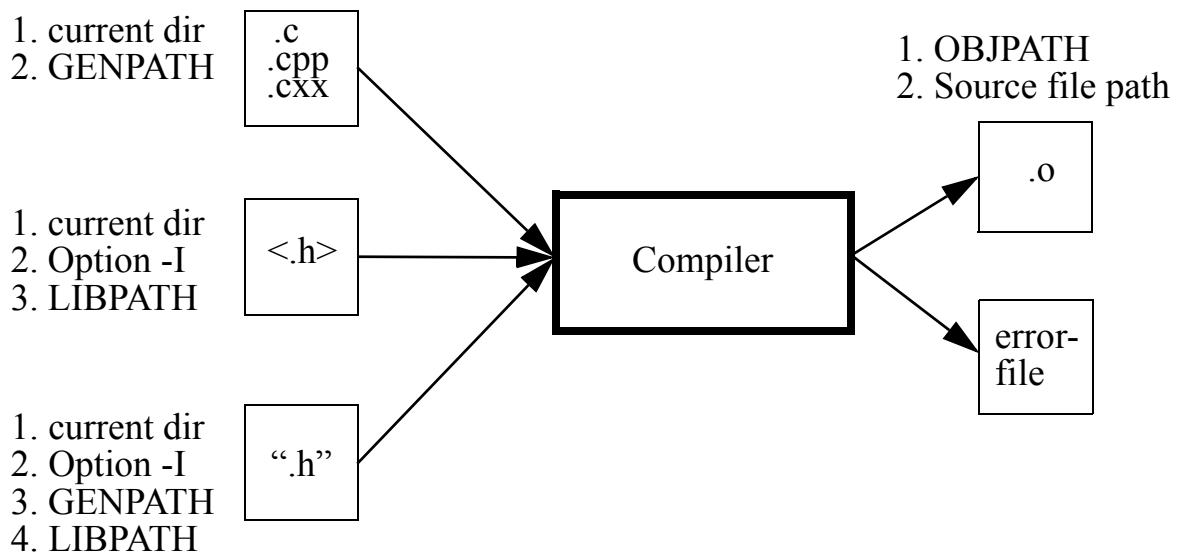
USERNAME=The Master

See also

[Environment variable COPYRIGHT](#)
[Environment variable INCLUDETIME](#)

Files

The following figure shows how file processing occurs with the Compiler:



Input Files

Source Files

The front end takes any file as input. It does not require the file name to have a special extension. However, it is suggested that all your source file names have the .c extension, and all header files the .h extension. The C++ files normally have the extension .cpp or .cxx. Source files are searched first in the [current directory](#) and then in the [GENPATH](#) directory.

Include File

The search for include files is governed by two environment variables: [GENPATH](#) and [LIBRARYPATH](#). Include files that are included using double quotes as in

```
#include "test.h"
```

are searched first in the current directory, then in the directory specified by the [-I option](#), then in the directories given in the environment variable [GENPATH](#), and finally in those listed in [LIBPATH](#) or [LIBRARYPATH](#). The current directory is set using the IDE, the Program Manager, or the [DEFAULTDIR](#) environment variable.

Include files that are included using angular brackets as in

```
#include <stdio.h>
```

are searched for first in the current directory, then in the directory specified by the -I option, and then in the directories given in [LIBPATH](#) or [LIBRARYPATH](#). The current directory is set using the IDE, the Program Manager, or the environment variable [DEFAULTDIR](#).

Output Files

Object Files

After successful compilation, the Compiler generates an object file containing the target code as well as some debugging information. This file is written to the directory listed in the [OBJPATH](#) environment variable. If that variable contains more than one path, the object file is written in the first listed directory. If this variable is not set, the object file is written in the directory the source file was found. Object files always get the extension .o.

Error Listing

If the Compiler detects any errors, it does not create an object file. Rather, it creates an error listing file named ERR.TXT. This file is generated in the directory the source file was found (also see *Environment*, [Environment Variable ERRORFILE](#)).

If the Compiler's window is open, it displays the full path of all header files read. After successful compilation the number of code bytes generated and the number of global objects written to the object file are also displayed.

If the Compiler is started from an IDE (with '%f' given on the command line) or Codewright (with '%b%e' given on the command line), this error file is not produced. Instead, it writes the error messages in a special format in a file called EDOOUT using the Microsoft format by default. You may use the CodeWrights's *Find Next Error* command to display both the error positions and the error messages.

Interactive Mode (Compiler Window Open)

If ERRORFILE is set, the Compiler creates a message file named as specified in this environment variable.

If ERRORFILE is not set, a default file named ERR.TXT is generated in the current directory.

Batch Mode (Compiler Window Not Open)

If ERRORFILE is set, the Compiler creates a message file named as specified in this environment variable.

If ERRORFILE is not set, a default file named EDOOUT is generated in the current directory.

Using the Compiler

Files

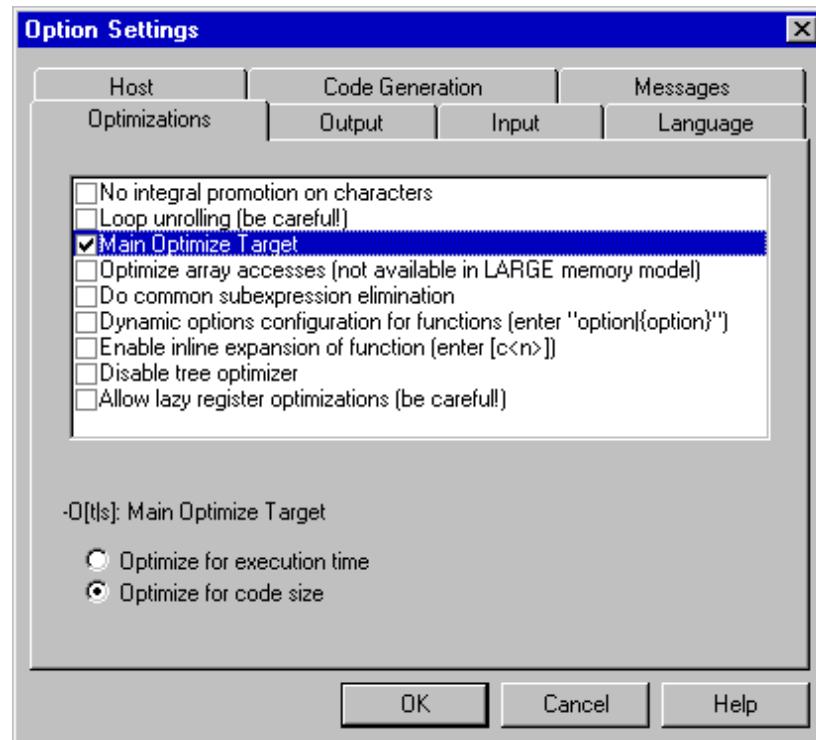
Compiler Options

The Compiler provides a number of Compiler options that control the Compiler's operation. Options consist of a minus/dash ('-'), followed by one or more letters or digits. Anything not starting with a dash/minus is the name of a source file to be compiled. You can specify Compiler options on the command line or in the COMPOPTIONS variable. Each Compiler option is specified only once per compilation.

Command line options are not case sensitive, e.g. "-Li" is the same as "-li".

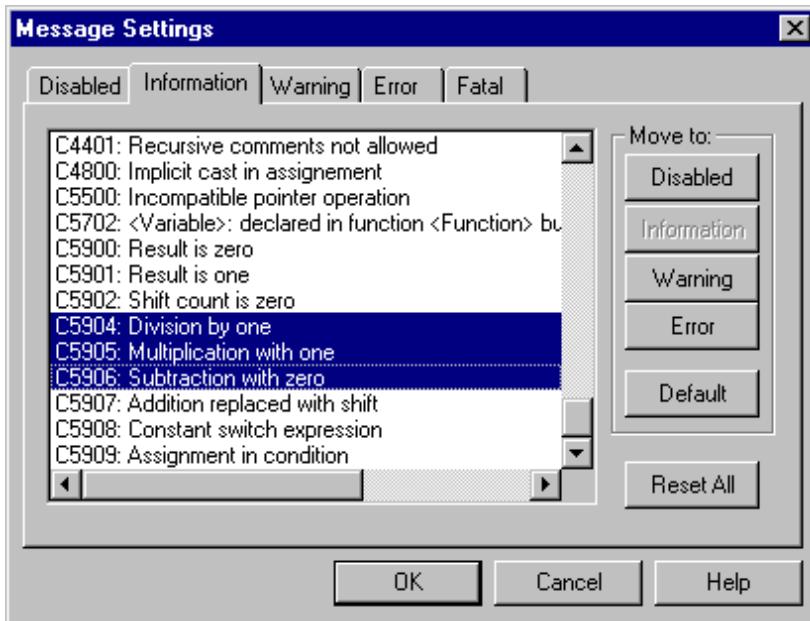
NOTE It is not possible to coalesce options in different groups, e.g. "-Cc -Li" *cannot* be abbreviated by the terms "-Cci" or "-CcLi"!

Another way to set the Compiler options is to use the GUI.



NOTE Do not use the COMPOPTIONS environment variable if the GUI is used. The Compiler stores the options in the project.ini, not in the default.env.

The dialog, shown below, may also be used to move messages (options -Wmsg).



Option Recommendation

Depending on the compiled sources, each Compiler optimization may have its advantages or disadvantages. The following is recommended:

- When using the HIWARE Object File Format: [-Cc](#) (allocate constant objects into ROM), remembering to specify ROM_VAR in the Linker parameter file
- [-Wpd](#) (error for implicit parameter declaration)
- Enable register optimizations (-Or) whenever available or possible

By default, most optimizations are enabled in the Compiler. If they cause problems in your code (e.g. they make the code hard to debug), switch them off (the options usually have the prefix -On). Candidates for such optimizations are peephole optimizations.

Some optimizations may produce more code for some functions than for others (e.g. inlining ([-Oi](#)), loop unrolling ([-Cu](#))). Try those options to get the best result for each.

To acquire the best results for each function, compile each module with the [-OdocF](#). An example for the this option is [-OdocF="-Or"](#).

Compiler Option Details

Option Groups

Compiler options are grouped by:

- HOST
- LANGUAGE
- OPTIMIZATIONS
- CODE GENERATION
- OUTPUT
- INPUT
- TARGET
- MESSAGES
- VARIOUS.

See [Table 1.12](#).

A special group is the STARTUP group: The options in this group cannot be specified interactively; they can only be specified on the command line to start the tool.

Table 1.12 Compiler Option Groups

Group	Description
HOST	Lists options related to the host
LANGUAGE	Lists options related to the programming language (ANSI C, C++, ...)
OPTIMIZATIONS	Lists optimization options
OUTPUT	Lists options related to the output files generation (which kind of file should be generated)
INPUT	Lists options related to the input file
CODE GENERATION	Lists options related to code generation (memory models, float format, ...)
TARGET	Lists options related to the target processor
MESSAGES	Lists options controlling the generation of error messages
VARIOUS	Lists various options
STARTUP	Options which only are specified on tool startup

The group corresponds to the property sheets of the graphical option settings.

NOTE	Not all command line options are accessible through the property sheets as they have a special graphical setting (e.g. the option to set the type sizes).
-------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------

Option Scopes

Each option has also a scope. See [Table 1.13](#)

Table 1.13 Option Scopes

Scope	Description
Application	The option has to be set for all files (Compilation Units) of an application. A typical example is an option to set the memory model. Mixing object files will have unpredictable results.
Compilation Unit	This option is set for each compilation unit for an application differently. Mixing objects in an application is possible.
Function	The option may be set for each function differently. Such an option may be used with the option: _OdocF .
None	The option scope is not related to a specific code part. A typical example are the options for the message management.

The available options are arranged into different groups. A sheet is available for each of these groups. The content of the list box depends on the selected sheets.

Option Detail Description

The remainder of this section describes each of the Compiler options available for the Compiler. The options are listed in alphabetical order. Each is divided into several sections listed in [Table 1.14](#).

Table 1.14 Compiler Option—Documentation Topics

Topic	Description
Group	HOST, LANGUAGE, OPTIMIZATIONS, OUTPUT, INPUT, CODE GENERATION, MESSAGE or VARIOUS.
Scope	Application, Compilation Unit, Function or None
Syntax	Specifies the syntax of the option in an EBNF format

Table 1.14 Compiler Option—Documentation Topics

Topic	Description
Arguments	Describes and lists optional and required arguments for the option
Default	Shows the default setting for the option
Defines	List of defines related to the Compiler option
Pragma	List of pragmas related to the Compiler option
Description	Provides a detailed description of the option and how to use it
Example	Gives an example of usage, and effects of the option where possible. Compiler settings, source code and/or Linker PRM files are displayed where applicable. The examples shows an entry in the default.env for a PC.
See also	Names related options

Using Special Modifiers

With some options, it is possible to use special modifiers. However, some modifiers may not make sense for all options. This section describes those modifiers.

[Table 1.15](#) lists the supported modifiers.

Table 1.15 Compiler Option Modifiers

Modifier	Description
%p	Path including file separator
%N	File name in strict 8.3 format
%n	File name without extension
%E	Extension in strict 8.3 format
%e	Extension
%f	Path + file name without extension
%"	A double quote ("") if the file name, the path or the extension contains a space
%'	A single quote ('') if the file name, the path or the extension contains a space
%(ENV)	Replaces it with the contents of an environment variable
%%	Generates a single '%'

Examples:

For your examples it is assumed that the actual file name (base file name for the modifiers) is:

c:\Metrowerks\my demo\TheWholeThing.myExt

%p gives the path only with a file separator:

c:\Metrowerks\my demo\

%N results in the file name in 8.3 format, that is the name with only 8 characters:

TheWhole

%n returns just the file name without extension:

TheWholeThing

%E gives the extension in 8.3 format, that is the extension with only 3 characters:

myE

%e is used for the whole extension:

myExt

%f gives the path plus the file name:

c:\Metrowerks\my demo\TheWholeThing

Because the path contains a space, using %" or %' is recommended: Thus %"%"% gives:

"c:\Metrowerks\my demo\TheWholeThing"

where %'%' gives:

'c:\Metrowerks\my demo\TheWholeThing'

Using %(envVariable) an environment variable may be used. A file separator following after %(envVariable) is ignored if the environment variable is empty or does not exist. In other words, the %(TEXTPATH)\myfile.txt is replaced with

c:\Metrowerks\txt\myfile.txt

if TEXTPATH is set to

TEXTPATH=c:\Metrowerks\txt

But is set to

myfile.txt

if TEXTPATH does not exist or is empty.

A %% may be used to print a percent sign. Using %e%% gives:

myExt%

-!: File Names to DOS Length

Group

INPUT

Scope

Compilation Unit

Syntax

"-!"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option, named *cut*, is very useful when compiling files copied from an MS-DOS file system. File names are clipped to DOS length (8 characters).

Example

- !

This truncates the following include directive

```
#include "mylongfilename.h"
```

to

```
#include "mylongfi.h"
```

See also

None

-AddIncl: Additional Include File

Group

INPUT

Scope

Compilation Unit

Syntax

"-AddIncl<fileName>"

Arguments

<fileName>: name of file to be included

Default

None

Defines

None

Pragmas

None

Description

The specified file is included at the beginning of the compilation unit. It has the same effect as it would if written at the beginning of the compilation unit using double quotes (".."):

```
#include "my headerfile.h"
```

Example

```
-AddIncl"my headerfile.h"
```

See also

[Option -I](#)

-A_ns_i: Strict ANSI

Group

LANGUAGE

Scope

Function

Syntax

"-A_ns_i"

Arguments

None

Default

None

Defines

__STDC__

Pragmas

None

Description

The `-ANSI` option forces the Compiler to follow strict ANSI C language conversions. When `-ANSI` is specified, all non ANSI-compliant keywords (e.g. `asm`, `__far` and `__near`) are not accepted by the Compiler, and the Compiler generates an error.

The compiler also does not allow C++ style comments (started with `//`). To allow C++ comments even with `-Ansi` set, the [option `-Cppc`](#) must be set.

The `asm` keyword is also not allowed if `-Ansi` is set. To use inline assembly even with `-Ansi` set, you have to use `__asm` instead the `asm` keyword.

The Compiler defines `__STDC__` as 1 if this option is set, and as 0 if this option is not set.

Example

-ANSI

See also

[Option -Cppc](#)

-BfaB: Bit Field Byte Allocation

Group

CODE GENERATION

Scope

Function

Syntax

"-BfaB" ("MS" | "LS")

Arguments

"MS": Most significant bit in byte first (left to right)

"LS": Least significant bit in byte first (right to left)

Default

XGATE: -BfaBLS

Defines

__BITFIELD_MSWORD_FIRST__
__BITFIELD_LSWORD_FIRST__
__BITFIELD_MSBYTE_FIRST__
__BITFIELD_LSBYTE_FIRST__
__BITFIELD_MSBIT_FIRST__
__BITFIELD_LSBIT_FIRST__

Pragmas

None

Description

Normally, bits in byte bitfields are allocated from the least significant bit to the most significant bit. This produces less code overhead if a byte bitfield is allocated only partially.

Example:

```
struct {unsigned char b: 3; } B;  
// by default the 3 least significant bits are used
```

This allows just a mask operation without any shift to access the bit field.

To change this allocation order, use the following option, shown in the example below.

Example

```
struct {  
    char b1:1;  
    char b2:1;  
    char b3:1;  
    char b4:1;  
    char b5:1;  
} myBitfield;
```

7	0

b1 b2 b3 b4 b5 ##### (-BfaBMS)	

7	0

##### b5 b4 b3 b2 b1 (-BfaBLS)	

Example

-BfaBMS

See also

[Bit Field Allocation Defines](#)

-BfaGapLimitBits: Bit Field Gap Limit

Group

CODE GENERATION

Scope

Function

Syntax

"-BfaGapLimitBits" <number>

Arguments

<number>: positive number specifying the maximum number of bits for a gap

XGATE: 0

Defines

None

Pragmas

None

Description

The bit field allocation tries to avoid crossing a byte boundary whenever possible. To achieve optimized accesses, the compiler may insert some padding/gap bits to reach this. This option enables you to affect the maximum number of gap bits allowed.

Example

```
struct {
    unsigned char a: 7;
    unsigned char b: 5;
    unsigned char c: 4;
} B;
```

In the above example, it is assumed that you have specified a 3-bit gap. The compiler allocates the struct B with 3 bytes. First the compiler allocates the 7 bits of a. Then the compiler tries to allocate the 5 bits of b, but this would cross a byte boundary. Because the gap of 1 bit is smaller than the specified gap of 3 bits, the b is allocated in the next byte. Then allocation starts for c. After allocation of b, there are 3 bits left. Because the gap is 3 bits, c is allocated in the next byte. If the maximum gap size is specified to 0, all bits would be allocated in two bytes.

Example

```
-BfaGapLimitBits3
```

See also

[Bit Field Allocation Defines](#)

-BfaTSR: Bit Field Type Size Reduction

Group

CODE GENERATION

Scope

Function

Syntax

"-BfaTSR" ("ON" | "OFF")

Arguments

"ON": Enable Type Size Reduction

"OFF": Disable Type Size Reduction

XGATE: -BfaTSRon

Defines

```
__BITFIELD_TYPE_SIZE_REDUCTION__
__BITFIELD_NO_TYPE_SIZE_REDUCTION__
```

Pragmas

None

Description

This option is configurable if the compiler uses type size reduction, or not, for bit fields. Type size reduction means that the compiler can reduce the type of an int bit field to a char bit field if it fits into a character. This allows the compiler to allocate memory only for one byte instead of for an integer.

Example

```
struct{
    long b1:4;
    long b2:4;
} myBitfield;
```

31 7 3 0

```
| #####|b2|b1|-BfaTSRoff  
-----  
7     3   0  
-----  
|b2 | b1 | -BfaTSRon  
-----
```

Example

-BfaTSRon

See also

[Bit Field Type Reduction Defines](#)

-C++ (-C++f, -C++e, -C++c): C++ Support

C++

Group

LANGUAGE

Scope

Compilation Unit

Syntax

"-C++" ("f"|"e"|"c")

Arguments

"f": Full ANSI Draft C++ support

"e": Embedded C++ support (EC++)

"c": compactC++ support (cC++)

Default

None

Defines

__cplusplus

Pragmas

None

Description

With this option enabled, the Compiler behaves as a C++ Compiler. You can choose between 3 different types of C++:

- Full ANSI Draft C++ supports the whole C++ language.
- Embedded C++ (EC++) supports a constant subset of the C++ language. EC++ does not support inefficient stuff like templates, multiple inheritance, virtual base classes and exception handling.
- compactC++ (cC++) supports a configurable subset of the C++ language. You can configure this subset with the option [-Cn](#).

If the option is not set, the Compiler behaves as an ANSI-C Compiler.

If the option is enabled and the source file name extension is * . c, the Compiler behaves as a C++ Compiler.

If the option is not set, but the source-filename extension is * . cpp or * . cxx, the Compiler behaves as if the -C++f option were set.

Example

```
COMPOPTIONS=-C++f
```

See also

[C++ Front End](#)
[Option -Cn](#)

-Cc: Allocate Constant Objects into ROM

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-Cc"

Arguments

None

Default

None

Defines

None

Pragmas

[#pragma INTO_ROM](#)

Description

In the HIWARE Object File Format, variables declared as `const` are treated just like any other variable, unless the command line option "`-Cc`" has been given. In that circumstance, the `const` objects are put into segment `ROM_VAR`, which is then assigned to a ROM section in the Linker parameter file (please see the *Linker manual*).

For objects allocated into a read-only section, the Linker prepares no initialization. The startup code does not have to copy the constant data.

You may also put variables into segment `ROM_VAR` by using the segment pragma (please see the *Linker manual*).

With the `#pragma CONST_SECTION` for constant segment allocation, variables declared as `const` are allocated in this segment.

If the current data segment is not the default segment, `const` objects in that user-defined segment are not allocated in the segment `ROM_VAR` but remain in the segment defined by the user. If that data segment happens to contain *only* `const` objects, it may be allocated in a ROM memory section (refer to the *Linker* manual for more information).

NOTE This option is useful only for HIWARE object file formats. Constants are allocated into the ELF section “`.rodata`” in the ELF/DWARF object file format.

NOTE The Compiler uses the default addressing mode for the constants specified by the memory model.

Example

```
-Cc  
SECTIONS  
    MY_ROM    READ_ONLY        0x1000 TO 0x2000  
PLACEMENT  
    DEFAULT_ROM, ROM_VAR    INTO MY_ROM
```

See also

[Segmentation](#)
Linker Manual
[Option -F2](#)
[#pragma INTO_ROM](#)

-Ccx: Cosmic compatibility mode for space modifiers and interrupt handlers

Group

LANGUAGE

Scope

Compilation Unit

Syntax

"-Ccx"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option allows Cosmic style @near, @far and @tiny space modifiers as well as @interrupt in your C code. The option -ANSI must be switched off. It is not necessary to remove the Cosmic space modifiers from your application code. There is no need to place the objects to sections addressable by the Cosmic space modifiers.

Following is done when a Cosmic modifier is parsed:

- The objects declared with the space modifier are always allocated in a special Cosmic compatibility (_cx...) section (regardless which section pragma is set) depending on the space modifier, on the const qualifier or if it is a function or a variable:

Table 1.16 Cosmic Modifier Handling

Definition	Placement to _cx section
@tiny int my_var	_CX_DATA_TINY
@near int my_var	_CX_DATA_NEAR
@far int my_var	_CX_DATA_FAR
const @tiny int my_cvar	_CX_CONST_TINY
const @near int my_cvar	_CX_CONST_NEAR
const @far int my_cvar	_CX_CONST_FAR
@tiny void my_fun(void)	_CX_CODE_TINY
@near void my_fun(void)	_CX_CODE_NEAR
@far void my_fun(void)	_CX_CODE_FAR
@interrupt void my_fun(void)	_CX_CODE_INTERRUPT

- Space modifiers on the left hand side of a pointer declaration specify the pointer type and pointer size, depending on the target.

See the example below for a prm file about how to place the sections mentioned in the table above.

For further information about porting applications from Cosmic to CodeWarrior please refer to the technical note TN 234.

The following table gives an overview how space modifiers are mapped for XGATE:

Table 1.17 Cosmic Space Modifier Mapping for XGATE

Definition	Keyword Mapping
@tiny	ignored
@near	ignored
@far	ignored

Example

-Ccx

See [Listing 1.1](#)

Listing 1.1 Cosmic Space Modifiers

```
volatile @tiny char tiny_ch;
extern @far const int table[100];
static @tiny char * @near ptr_tab[10];
typedef @far int (*@far funptr)(void);
funptr my_fun; /* banked and __far calling conv. */

char @tiny *tptr = &tiny_ch;
char @far *fptr = (char @far *)&tiny_ch;
```

Example for a prm file:

```
(16 and 24 bit addressable ROM;
 8, 16 and 24 bit addressable RAM)
```

```
SEGMENTS
  MY_ROM    READ_ONLY      0x2000 TO 0x7FFF;
  MY_BANK   READ_ONLY      0x508000 TO 0x50BFFF;
  MY_ZP     READ_WRITE     0xC0 TO 0xFF;
  MY_RAM    READ_WRITE     0xC000 TO 0xCFFF;
  MY_DBANK  READ_WRITE     0x108000 TO 0x10BFFF;

END
PLACEMENT
  DEFAULT_ROM, ROM_VAR,
  _CX_CODE_NEAR, _CX_CODE_TINY, _CX_CONST_TINY,
  _CX_CONST_NEAR           INTO MY_ROM;
  _CX_CODE_FAR, _CX_CONST_FAR  INTO MY_BANK;
  DEFAULT_RAM, _CX_DATA_NEAR   INTO MY_RAM;
  _CX_DATA_FAR             INTO MY_DBANK;
  _ZEROPAGE, _CX_DATA_TINY    INTO MY_ZP;
END
```

See also

Cosmic Manuals, Linker Manual, TN 234

-Ci: Tri- and Bigraph Support

Group

LANGUAGE

Scope

Function

Syntax

"-Ci"

Arguments

None

Default

None

Defines

__TRIGRAPHS__

Pragmas

None

Description

If certain tokens are not available on your keyboard, they are replaced with keywords as shown in [Table 1.18](#).

Table 1.18 Keyword Alternatives for Unavailable Tokens

Bigraph		Trigraph		Additional Keyword	
<%	}	??=	#	and	&&
%>	}	?? /	\	and_eq	&=
<:	[?? '	^	bitand	&
:>]	?? ([bitor	

Table 1.18 Keyword Alternatives for Unavailable Tokens

Bigraph		Trigraph		Additional Keyword	
%:	#	??)]	compl	~
% : % :	##	?? !		not	!
		??<	{	or	
		??>	}	or_eq	=
		?? -	~	xor	^
				xor_eq	^ =
				not_eq	! =

NOTE	Additional keywords are not allowed as identifiers if this option is enabled.
-------------	-------------------------------------------------------------------------------

Example

-Ci

The example in [Listing 1.2](#) shows the use of trigraphs, bigraphs and the additional keywords with the corresponding ‘normal’ C-source.

Listing 1.2 Trigraphs, Bigraphs, and Additional Keywords

```

int Trigraphs(int argc, char * argv??(??)) ??<
    if (argc<1 ??!??! *argv??(1??)=='??/0') return 0;
    printf("Hello, %s??/n", argv??(1??));
??>

?:define TEST_NEW_THIS 5
?:define cat(a,b) a%:%:b
??:define arraycheck(a,b,c) a??(i??) ??!??! b??(i??)

int i;
int cat(a,b);
char a<:10:>;
char b<:10:>;

void Trigraph2(void) <%
    if (i and ab) <%

```

```
i and_eq TEST_NEW_THIS;
i = i bitand 0x03;
i = i bitor 0x8;
i = compl i;
i = not i;
%> else if (ab or i) <%
    i or_eq 0x5;
    i = i xor 0x12;
    i xor_eq 99;
%> else if (i not_eq 5) <%
    cat(a,b) = 5;
    if (a??(i??) || b[i])<%>
    if (arraycheck(a,b,i)) <%
        i = 0;
    %>
%>
%>

/* is the same as ... */
int Trigraphs(int argc, char * argv[]) {
    if (argc<1 || *argv[1]=='\0') return 0;
    printf("Hello, %s\n", argv[1]);
}

#define TEST_NEW_THIS 5
#define cat(a,b) a##b
#define arraycheck(a,b,c) a[i] || b[i]

int i;
int cat(a,b);
char a[10];
char b[10];

void Trigraph2(void){
    if (i && ab) {
        i &= TEST_NEW_THIS;
        i = i & 0x03;
        i = i | 0x8;
        i = ~i;
        i = !i;
    } else if (ab || i) {
        i |= 0x5;
        i = i ^ 0x12;
        i ^= 99;
```

Using the Compiler

Compiler Options

```
} else if (i != 5) {  
    cat(a,b) = 5;  
    if (a[i] || b[i]){}  
    if (arraycheck(a,b,i)) {  
        i = 0;  
    }  
}  
}
```

See also

None

-Cn: Disable compactC++ features

C++

Group

LANGUAGE

Scope

Compilation Unit

Syntax

"-Cn" ["=" {"Vf"|"Tpl"|"Ptm"|"Mih"|"Ctr"|"Cpr"}].

Arguments

"Vf": Do not allow virtual functions
"Tpl": Do not allow templates
"Ptm": Do not allow pointer to member
"Mih": Do not allow multiple inheritance and virtual base classes
"Ctr": Do not create compiler defined functions
"Cpr": Do not allow class parameters and class returns

Default

None

Defines

None

Pragmas

None

Description

If the [-C++c](#) option is enabled, you can disable following compactC++ features:

"Vf" Virtual functions are not allowed.
Avoid having virtual tables consuming a lot of memory.

"Tpl" Templates are not allowed.
Avoid having many generated functions performing similar operations.

"Ptm" Pointer to member not allowed.

Avoid having pointer-to-member objects consuming a lot of memory.

"Mih" Multiple inheritance is not allowed.

Avoid having complex class hierarchies.

Because virtual base classes are logical only when used with multiple inheritance, they are also not allowed.

"Ctr" The C++ Compiler can generate several kinds of functions, if necessary:

- Default Constructor,
- Copy Constructor,
- Destructor,
- Assignment operator.

With this option enabled, the Compiler does not create those functions. This is useful when compiling C sources with the C++ Compiler, assuming you do not want C structures to acquire member functions.

"Cpr" Class parameters and class returns are not allowed.

Avoid having overhead with Copy Constructor and Destructor calls at passing parameters, and passing return values of class type.

Example

`-C++c -Cn=Ctr`

See also

[Option -C++c](#)

[C++ Front End](#)

-Cni: No Integral Promotion

Group

OPTIMIZATIONS

Scope

Function

Syntax

"-Cni"

Arguments

None

Default

None

Defines

__CNI__

Pragmas

None

Description

Enhances code density of character operations by omitting integral promotion. This option enables a non ANSI-C compliant behavior.

In ANSI-C operations with data types, anything smaller than int must be promoted to int (integral promotion). With this rule, adding two unsigned character variables results in a zero-extension of each character operand, and then adding them back in as int operands. If the result must be stored back into a character, this integral promotion is not necessary. When this option is set, promotion is avoided where possible.

The code size may be decreased if this option is set, because operations may be performed on a character base instead of an integer base.

The **-Cni** option enhances character operation code density by omitting integral promotion.

Consider the following:

In most expressions, ANSI-C requires char type variables to be extended to the next larger type int, which is required to be at least 16-bit in size by the ANSI standard.

The **-Cni** option suppresses this ANSI-C behavior and thus allows 'characters' and 'character sized constants' to be used in expressions. This option does not conform to ANSI standards. Code compiled with this option is not portable.

The ANSI standard requires that 'old style declarations' of functions using the char parameter be extended to int. The **-Cni** option disables this extension saving additional RAM. See the example below.

Example

```
old_style_func (a, b, c)
    char a, b, c;
{
    ...
}
```

The space reserved for a, b, c is just one byte each, instead of two.

For expressions containing different types of variables, the following conversion rules apply:

If both variables are of type signed char, the expression is evaluated signed.

If one of two variables is of type unsigned char, the expression is evaluated unsigned, regardless of whether the other variable is of type signed or unsigned char.

If one operand is of another type than signed or unsigned char, the usual ANSI-C arithmetic conversions are applied.

If constants are in the character range, they are treated as characters. Remember that the char type is signed and applies to the constants -128 – 127. All constants greater than 127, i.e. 128, 129 ... are treated as integer. If you want them treated as characters, they must be casted.

Example

```
signed char a, b;
if (a > b * (signed char) - 129)
```

NOTE This option is ignored with the `-ANSI` Compiler switch active.

NOTE With this option set, the code that is generated does not conform to the ANSI standard. In other words: the code generated is wrong if you apply the ANSI standard as reference. Using this option is not recommended in most cases.

Example

`-Cni`

See also

None

-Cppc: C++ Comments in ANSI-C

Group

LANGUAGE

Scope

Function

Syntax

"-Cppc"

Arguments

None

Default

By default, the Compiler does not allow C++ comments if [-Ansi](#) is set.

Defines

None

Pragmas

None

Description

The `-Ansi` option forces the compiler to conform to the ANSI-C standard. Because a strict ANSI-C compiler rejects any C++ comments (started with `//`), this option may be used to allow C++ comments.

Example

`-Cppc`

This allows the following code to be compiled with the `-Ansi` option set:

```
void foo(void) // this is a c++ comment
```

See also

[Option -Ansi](#)

-Cq: Propagate const and volatile Qualifiers for structs

Group

LANGUAGE

Scope

Application

Syntax

"-Cq"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option propagates const and volatile qualifiers for structures. That means, if all members of a structure are constant, the structure itself is constant as well. The same happens with the volatile qualifier. If the structure is declared as constant/volatile, all members are constant/volatile. Consider the following example:

```
struct {
    const field;
} s1, s2;

void foo(void) {
    s1 = s2; // struct copy
    s1.field = 3; // error: modifiable lvalue expected
}
```

In the above example, the field in the struct is constant, but not the struct itself. Thus the struct copy ‘s1 = s2’ is legal, even if the field of the struct is constant. But, a write access to the struct field causes an error message. Using the -Cq option propagates the qualification (const) of the fields to the whole struct or array. In the above example, the struct copy would cause an error message.

Example

-Cq

See also

None

**-CsIni0: Assume SP register is zero initialized at
thread start**

XGATE

Group

CODE GENERATION

Scope

Function

Syntax

"-CsIni0"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The reference definition of the XGATE architecture does specify that the registers R2 up to R7 are undefined at thread start. However the first implementation of the XGATE does actually always zero initialize these registers. As this behavior is not defined, the compiler does provide this option to generate code which does benefit from this or to generate code which does work even if the R7 register is not initially 0.

If you are uncertain about using this option, the safe default is to not specify it.

The code generated with this option is one instruction smaller for all interrupt functions which do not allocate space on the stack and it also only helps is the low byte of the stack top is 0.

Example

`-Cstv=0xD000 -CsIni0`

See also

None

-Cstv: Initialize Stack

XGATE

Group

CODE GENERATION

Scope

Function

Syntax

"-Cstv=<(0x)n>"

Arguments

Address of first byte above the XGATE stack.

Default

None

Defines

None

Pragmas

None

Description

With this option, the address of the stack for the XGATE is passed to the compiler. The compiler will load the passed address in any interrupt function if the stack is used.

The address should specify the first byte which is no longer used. Using 0xD000 will cause that the topmost word of the stack is written to 0xFFE..0xFFFF.

Example

-Cstv=0xD000

See also

None

-CswMaxLF: Maximum Load Factor for Switch Tables

Group

CODE GENERATION

Scope

Function

Syntax

"-CswMaxLF<number>"

Arguments

<number>: a number in the range of 0 – 100 denoting the maximum load factor.

Default

Back End dependent.

Defines

None

Pragmas

None

Description

Allows changing the default strategy of the Compiler to use tables for switch statements.

NOTE This option is only available if the compiler supports switch tables.

Normally the Compiler uses a table for switches with more than about 8 labels if the table is filled between 80% (minimum load factor of 80) and 100% (maximum load factor of 100). If there are not enough labels for a table or the table is not filled, a branch tree is generated (tree of if-else-if-else). This branch tree is like an ‘unrolled’ binary search in a table which quickly evaluates the associated label for a switch expression.

Using a branch tree instead of a table improves code execution speed, but may increase code size. Additionally, because the branch tree itself uses no special runtime routine for switch expression evaluation, debugging may be more seamless.

Specifying a load factor means that tables are generated in specific ‘fuel’ status:

```
switch(i) {  
    case 0: ...  
    case 1: ...  
    case 2: ...  
    case 3: ...  
    case 4: ...  
    // case 5: ...  
    case 6: ...  
    case 7: ...  
    case 8: ...  
    case 9: ...  
    default  
}
```

The above table is filled to 90% (labels for ‘0’ to ‘9’, except for ‘5’). Assumed that the minimum load factor is set to 50% and setting the maximum load factor for the above case to 80%, a branch tree is generated instead a table. But setting the maximum load factor to 95% will produce a table.

To guarantee that tables are generated for switches with full tables only, set the table minimum and maximum load factors to 100:

-CswMinLF100 -CswMaxLF100.

Example

-CswMaxLF50

See also

[Option -CswMinLB](#)
[Option -CswMinSLB](#)
[Option -CswMinLF](#)

-CswMinLB: Minimum Number of Labels for Switch Tables

Group

CODE GENERATION

Scope

Function

Syntax

"-CswMinLB<number>"

Arguments

<number>: a positive number denoting the number of labels.

Default

Back End dependent

Defines

None

Pragmas

None

Description

This option allows changing the default strategy of the Compiler using tables for switch statements.

NOTE This option is only available if the compiler supports switch tables.

Normally the Compiler uses a table for switches with more than about 8 labels (case entries) (actually this number is highly backend dependent). If there are not enough labels for a table, a branch tree is generated (tree of if-else-if-else). This branch tree is like an ‘unrolled’ binary search in a table which evaluates very fast the associated label for a switch expression.

Using a branch tree instead of a table may increases the code execution speed, but it probably increase the code size. Additionally because the branch tree itself uses no special runtime routine for switch expression evaluation, debugging may be much easier.

To disable any tables for switch statements, just set the minimum number of labels needed for a table to a high value (e.g. 9999):

-CswMinLB9999 -CswMinSLB9999.

When disabling simple tables it usually makes sense also to disable search tables with option -CswMinSLB.

Example

-CswMinLB5

See also

[Option -CswMinLF](#)

[Option -CswMinSB](#)

[Option -CswMaxLF](#)

-CswMinLF: Minimum Load Factor for Switch Tables

Group

CODE GENERATION

Scope

Function

Syntax

"-CswMinLF<number>".

Arguments

<number>: a number in the range of 0 – 100 denoting the minimum load factor

Default

Back End dependent

Defines

None

Pragmas

None

Description

Allows the Compiler to use tables for switch statements.

NOTE	This option is only available if the compiler supports switch tables.
-------------	-----------------------------------------------------------------------

Normally the Compiler uses a table for switches with more than about 8 labels and if the table is filled between 80% (minimum load factor of 80) and 100% (maximum load factor of 100). If there are not enough labels for a table or the table is not filled, a branch tree is generated (tree of if-else-if-else). This branch tree is like an ‘unrolled’ binary search in a table which quickly evaluates the associated label for a switch expression.

Using a branch tree instead of a table improves code execution speed, but may increase code size. Additionally, because the branch tree itself uses no special runtime routine for switch expression evaluation, debugging is more seamless.

Specifying a load factor means that tables are generated in specific ‘fuel’ status:

```
switch(i) {  
    case 0: ...  
    case 1: ...  
    case 2: ...  
    case 3: ...  
    case 4: ...  
    // case 5: ...  
    case 6: ...  
    case 7: ...  
    case 8: ...  
    case 9: ...  
    default  
}
```

The above table is filled to 90% (labels for ‘0’ to ‘9’, except for ‘5’). Assuming that the maximum load factor is set to 100% and the minimum load factor for the above case is set to 90%, this still generates a table. But setting the minimum load factor to 95% produces a branch tree.

To guarantee that tables are generated for switches with full tables only, set the minimum and maximum table load factors to 100: -CswMinLF100
-CswMaxLF100.

Example

-CswMinLF50

See also

[Option -CswMinLB](#)
[Option -CswMinSLB](#)
[Option -CswMaxLF](#)

-CswMinSLB: Minimum Number of Labels for Search Switch Tables

Group

CODE GENERATION

Scope

Function

Syntax

"-CswMinSLB<number>"

Arguments

<number>: a positive number denoting the number of labels

Default

Back End dependent

Defines

None

Pragmas

None

Description

Allows the Compiler to use tables for switch statements.

NOTE	This option is only available if the compiler supports search tables.
-------------	-----------------------------------------------------------------------

Switch tables are implemented in different ways. When almost all case entries in some range are given, a table containing only branch targets is used. Using such a dense table is efficient because only the correct entry is accessed. When large holes exist in some areas, a table form can still be used.

But now the case entry and its corresponding branch target are encoded in the table. This is called a search table. A complex runtime routine must be used to

access a search table. This routine checks all entries until it finds the matching one. Search tables execute slowly.

Using a search table improves code density, but the execution time increases. Every time an entry in a search table must be found, all previous entries must be checked first. For a dense table, the right offset is computed and accessed. Additionally note that all backends implement search tables (if at all) by using a complex runtime routine. This may make debugging more complex.

To disable search tables for switch statements, set the minimum number of labels needed for a table to a high value (e.g. 9999): -CswMinSLB9999.

Example

-CswMinSLB9999

See also

[Option -CswMinLB](#)
[Option -CswMinLF](#)
[Option -CswMaxLF](#)

-Cu: Loop Unrolling

Group

OPTIMIZATIONS

Scope

Function

Syntax

"-Cu ["=i" <number>]"

Arguments

<number>: number of iterations for unrolling, between 0 and 1024

Default

None

Defines

None

Pragmas

[#pragma LOOP_UNROLL](#)
[#pragma NO_LOOP_UNROLL](#)

Description

Enables loop unrolling with the following restrictions:

- Only simple `for` statements are unrolled, e.g.
`for (i=0; i<10; i++)`
- Initialization and test of the loop counter must be done with a constant.
- Only `<`, `>`, `<=`, `>=` are permitted in a condition.
- Only `++` or `--` are allowed for the loop variable increment or decrement.
- The loop counter must be integral.
- No change of the loop counter is allowed within the loop.

- The loop counter must not be used on the left side of an assignment.
- No address operator (&) is allowed on the loop counter within the loop.
- Only small loops are unrolled:
- Loops with few statements within the loop.
- Loops with fewer than 16 increments or decrements of the loop counter.
The bound may be changed with the optional argument "=i" <number>. The -Cu=i20 option unrolls loops with a maximum of 20 iterations.

Example

```
-Cu  
int i, j;  
  
j = 0;  
for (i=0; i<3; i++) {  
    j += i;  
}
```

With the Compiler option '-Cu' given, the Compiler issues an information message '*Unrolling loop*' and transforms this loop into:

```
j += 1;  
j += 2;  
i = 3;
```

The Compiler also transforms some special loops, i.e. loops with a constant condition or loops with only one pass:

Example for a loop with a constant condition:

```
for (i=1; i>3; i++) {  
    j += i;  
}
```

The Compiler issues an information message '*Constant condition found, removing loop*' and transforms the loop into a simple assignment

```
i=1;
```

because the loop body is never executed.

Example for a loop with only one pass:

```
for (i=1; i<2; i++) {  
    j += i;  
}
```

The Compiler issues a warning '*Unrolling loop*' and transforms the `for` loop into

```
j += 1;  
i = 2;
```

because the loop body is executed only once.

See also

None

-Cx: No Code Generation

Group

CODE GENERATION

Scope

Compilation Unit

Syntax

"-Cx"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The -Cx Compiler option disables the code generation process of the Compiler. No object code is generated, though the Compiler does perform a syntactical check of the source code. This allows a quick test if the Compiler accepts the source without errors.

Example

-Cx

See also

None

-D: Macro Definition

Group

LANGUAGE

Scope

Compilation Unit

Syntax

"-D" <identifier> ["=" <value>]

Arguments

<identifier>: identifier to be defined

<value>: value for <identifier>, anything except "-" and blank

Default

None

Defines

None

Pragmas

None

Description

The Compiler allows the definition of a macro on the command line. The effect is the same as having a `#define` directive at the very beginning of the source file.

Example

`-DDEBUG=0`

This is the same as writing

```
#define DEBUG 0
```

in the source file.

If you need strings with blanks in your macro definition, there are two ways: escape sequence or double quotes:

```
-dPath="Path\40with\40spaces"  
-d "Path= "Path with spaces" "
```

NOTE Blanks are *not* allowed after the "-D" option – the first blank terminates this option. Also, macro parameters are not supported.

See also

None

-Ec: Conversion from 'const T*' to 'T*'

Group

LANGUAGE

Scope

Function

Syntax

"-Ec"

Arguments

None

Default

None

Description

If this non ANSI compliant extension is enabled, a pointer to a constant type is treated like a pointer to the non-constant equivalent of the type. Earlier Compilers did not check a store to a constant object through a pointer. This option is useful if some older source has to be compiled.

Examples

```
void f() {
    int *i;
    const int *j;
    i=j; /* C++ illegal, but with -ec ok! */
}

struct A {
    int i;
};

void g() {
    const struct A *a;
    a->i=3; /* ANSI-C/C++ illegal, but with -Ec ok! */
}
```

```
void h() {
    const int *i;
    *i=23; /* ANSI-C/C++ illegal, but with -ec ok! */
}
```

Defines

None

Pragmas

None

Example

-Ec

```
void foo(const int *p){
    *p = 0; // some Compiler does not issue an error
}
```

See also

None

-Eencrypt: Encrypt Files

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-Eencrypt" ["=" <filename>]

Arguments

<filename>: The name of the file to be generated

It may contain special modifiers (see [Using Special Modifiers](#)).

Default

The default file name is %f.e%e. A file named ‘foo.c’ creates an encrypted file named ‘foo.ec’.

Description

All files passed together with this option are encrypted using the given key with the option [-Ekey](#).

NOTE	This option is only available/operative with a license for the following feature: H1xxxx30 where xxxx is the feature number of the compiler for a specific target.
-------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------

Defines

None

Pragmas

None

Example

```
foo.c foo.h -Ekey1234567 -Eencrypt=%n.e%e
```

encrypts the file ‘foo.c’ using the key 1234567 to file ‘foo.ec’ and the file ‘foo.h’ to the file ‘foo.eh’.

The encrypted files foo.ec and foo.eh may be passed to a client. The client is able to compile the encrypted files without the key compiling the following file:

```
foo.ec
```

See also

[Option -Ekey](#)

-Ekey: Encryption Key

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-Ekey<keyNumber>"

Arguments

<keyNumber>

Default

The default encryption key is ‘0’. Using this default is not recommended.

Description

This option is used to encrypt files with the given key number (option [-E encrypt](#)).

NOTE

This option is only available/operative with a license for the following feature: HIxxxx30 where xxxx is the feature number of the compiler for a specific target.

Defines

None

Pragmas

None

Example

```
foo.c -Ekey1234567 -Eencrypt=%n.e%
```

encrypts the file ‘foo.c’ using the key 1234567

Using the Compiler

Compiler Options

See also

[Option -Eencrypt](#)

-Env: Set Environment Variable

Group

HOST

Scope

Compilation Unit

Syntax

"-Env" <Environment Variable> "=" <Variable Setting>

Arguments

<Environment Variable>: Environment variable to be set

<Variable Setting>: Setting of the environment variable

Default

None

Description

This option sets an environment variable. This environment variable may be used in the maker, or used to overwrite system environment variables.

Defines

None

Pragmas

None

Example

-EnvOBJPATH=\sources\obj

This is the same as

OBJPATH=\sources\obj

in the default.env.

Using the Compiler

Compiler Options

Use the following syntax to use an environment variable using file names with spaces:

```
-Env"OBJPATH=\program files"
```

See also

[Section “Environment”](#)

-F (-Fh, -F1, -F1o, -F2, -F2o,-F6, -F7): Object File Format

Group

OUTPUT

Scope

Application

Syntax

"-F" ("h" | "1" | "1o" | "2" | "2o" | "6" | "7")

Arguments

"h": HIWARE object file format
"1": ELF/DWARF 1.1 object file format
"1o": compatible ELF/DWARF 1.1 object file format
"2": ELF/DWARF 2.0 object file format
"2o": compatible ELF/DWARF 2.0 object file format
"6": strict HIWARE V2.6 object file format
"7": strict HIWARE V2.7 object file format

NOTE Not all object file formats may be available for a target.

Default

Freescale XGATE: -F2

Defines

__HIWARE_OBJECT_FILE_FORMAT__
__ELF_OBJECT_FILE_FORMAT__

Pragmas

None

Description

The Compiler writes the code and debugging info after compilation into an object file.

The Compiler uses a HIWARE-proprietary object file format when the -Fh, -F6 or -F7 options are set.

The HIWARE Object File Format (-Fh) has the following limitations:

- The type char is limited to a size of 1 byte.
- Symbolic debugging for enumerations is limited to 16-bit signed enumerations.
- No zero bytes in strings are allowed (zero byte marks the end of the string).

The HIWARE V2.7 Object File Format (option -F7) has some limitations:

- The type char is limited to a size of 1 byte.
- Enumerations are limited to a size of 2 bytes and have to be signed.
- No symbolic debugging for enumerations.
- The standard type short is encoded as int in the object file format.
- No zero bytes in strings allowed (zero byte marks the end of the string).

The Compiler produces an ELF/DWARF object file when the options -F1 or -F2 are set. This object file format may also be supported by other Compiler vendors.

In the Compiler ELF/DWARF 2.0 output, some constructs written in previous versions were not conforming to the ELF standard because the standard was not clear enough in this area. Because old versions of the simulator/debugger (V5.2 or earlier) are not able to load the corrected new format, the old behavior can still be produced by using "-f2o" instead of "-f2". Some old versions of the debugger (simulator/debugger V5.2 or earlier) generate a GPF when a new absolute file is loaded. If you want to use the older versions, use "-f2o" instead of "-f2". New versions of the debugger are able to load both formats correctly. Also, some older ELF/DWARF object file loaders from emulator vendors may require you to set the -F2o option.

The -F1o option is only supported if the target supports the ELF/DWARF 1.1 format. This option is only used with older debugger versions as a compatibility option. This option may be discontinued in the future. It is recommended you use -F1 instead.

Note that it is recommended to use the ELF/DWARF 2.0 format instead of the ELF/DWARF 1.1. The 2.0 format is much more generic. Additionally, it

supports multiple include files plus modifications of the basic generic types (e.g. floating point format). Debug information is also more robust.

Example

-Fh

See also

None

-H: Short Help

Group

VARIOUS

Scope

None

Syntax

"-H"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The -H option causes the Compiler to display a short list (i.e. help list) of available options within the Compiler window. Options are grouped into HOST, LANGUAGE, OPTIMIZATIONS, OUTPUT, INPUT, CODE GENERATION, MESSAGE and VARIOUS.

No other option or source file should be specified when the -H option is invoked.

Example

-H may produce following list:

```
INPUT:  
-!      Filenames ar clipped to DOS length  
-I      Include file path
```

VARIOUS:

- H Prints this list of options
- V Prints the Compiler version

See also

None

-I: Include File Path

Group

INPUT

Scope

Compilation Unit

Syntax

"-I" <path>

Arguments

<path>: path, terminated by a space or end-of-line

Default

None

Defines

None

Pragmas

None

Description

Allows you to set include paths in addition to the [LIBPATH](#), [LIBRARYPATH](#) and [GENPATH](#) environment variables. Paths specified with this option have precedence over includes in the current directory, and paths specified in [GENPATH](#), [LIBPATH](#) and [LIBRARYPATH](#).

Example

```
-I. -I..\\h -I\\src\\include
```

This directs the Compiler to search for header files first in the current directory (.), then relative from the current directory in '..\\h', and then in '\\src\\include'. If the file is not found, the search continues with [GENPATH](#), [LIBPATH](#) and [LIBRARYPATH](#) for header files in double quotes

(`#include "headerfile.h"`), and with [LIBPATH](#) and [LIBRARYPATH](#) for header files in angular brackets (`#include <stdio.h>`).

See also

[Section “Input Files”](#)

[Option -AddIncl](#)

[Environment variable LIBPATH](#)

[Environment variable LIBRARYPATH](#)

[Environment variable GENPATH](#)

-La: Generate Assembler Include File

Group

OUTPUT

Scope

Function

Syntax

"-La" ["=" <filename>]

Arguments

<filename>: The name of the file to be generated

It may contain special modifiers (see [Using Special Modifiers](#))

Default

No file created

Defines

None

Pragmas

None

Description

The -La option causes the Compiler to generate an assembler include file when the CREATE_ASM_LISTING pragma occurs. The name of the created file is specified by this option. If no name is specified, a default of "%f.inc" is taken. To put the file into the directory specified by the environment variable [TEXTPATH](#), use the option "-la=%n.inc". The %f option already contain the path of the source file. When %f is used, the generated file is in the same directory as the source file.

The content of all modifiers refers to the main input file and not to the actual header file. The main input file is the one specified on the command line.

Example

-La=asm.inc

See also

[pragma CREATE_ASM_LISTING](#)
[Generating an Assembler Include File](#)

-Lasm: Generate Listing File

Group

OUTPUT

Scope

Function

Syntax

"-Lasm" ["=" <filename>]

Arguments

<filename>: The name of the file to be generated.

It may contain special modifiers (see [Using Special Modifiers](#)).

Default

No file created.

Defines

None

Pragmas

None

Description

The -Lasm option causes the Compiler to generate an assembler listing file directly. All assembler generated instructions are also printed to this file. The name of the file is specified by this option. If no name is specified, a default of "%n.lst" is taken. The environment variable, [TEXTPATH](#), is used if the resulting file name contains no path information.

The syntax does not always conform with the inline assembler or the assembler syntax. Therefore, this option can only be used to review the generated code. It can not currently be used to generate a file for assembly.

Example

-Lasm=asm.lst

See also

[Option -Lasmc](#)

-Lasmc: Configure Listing File

Group

OUTPUT

Scope

Function

Syntax

"-Lasmc" ["=" {"a" | "c" | "i" | "s" | "h" | "p" | "e" | "v"}]

Arguments

- a: Do not write the address in front of every instruction
- c: Do not write the hex bytes of the instructions
- i: Do not write the decoded instructions
- s: Do not write the source code
- h: Do not write the function header
- p: Do not write the source prolog
- e: Do not write the source epilog
- v: Do not write the compiler version

Default

All printed together with the source

Defines

None

Pragmas

None

Description

The -Lasmc option configures the output format of the listing file generated with the [option -Lasm](#). The addresses, the hex bytes, and the instructions are selectively switched off.

The format of the listing file has layout shown below. The letters in brackets ([])) indicate which suboption may be used to switch it off:

```
[v] ANSI-C/cC++ Compiler V-5.0.1
[v]
[p]    1:
[p]    2: void foo(void) {
[h]
[h] Function: foo
[h] Source   : C:\Metrowerks\test.c
[h] Options  : -Lasm=%n.lst
[h]
[s]    3: }
[a] 0000 [c] 3d           [i] RTS
[e]    4:
[e]    5: // comments
[e]    6:
```

Example

-Lasmc=ac

See also

[Option -Lasm](#)

-Ldf: Log Predefined Defines to File

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-Ldf ["=" <file>]

Arguments

<file>: file name for the log file, default is ‘predef.h’.

Default

default <file> is ‘predef.h’.

Defines

None

Pragmas

None

Description

The -Ldf option causes the Compiler to generate a text file that contains a list of the Compiler-defined #define. The default file name is ‘predef.h’, but may be changed (e.g. -Ldf=”myfile.h”). The file is generated in the directory specified by the [TEXTPATH](#) environment variable. The defines written to this file depend on the actual Compiler option settings (e.g. type size settings, ANSI compliance, ...).

NOTE

The defines specified by the command line (option [-D](#)) are not included.

This option may be very useful for SQA. With this option it is possible to document every `#define` which was used to compile all sources.

NOTE	This option only has an effect if a file is compiled. This option is unusable if you are not compiling a file.
-------------	----------------------------------------------------------------------------------------------------------------

Example

-Ldf

This generates the file ‘predef.h’ with the following content:

```
/* resolved by preprocessor: __LINE__ */  
/* resolved by preprocessor: __FILE__ */  
/* resolved by preprocessor: __DATE__ */  
/* resolved by preprocessor: __TIME__ */  
#define __STDC__ 0  
#define __VERSION__ 5004  
#define __VERSION_STR__ "V-5.0.4"  
#define __SMALL__  
#define __PTR_SIZE_2__  
#define __BITFIELD_LSBIT_FIRST__  
#define __BITFIELD_MSBYTE_FIRST__  
...
```

See also

[Option -D](#)

-Li: List of Included Files

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-Li".

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The -Li option causes the Compiler to generate a text file which contains a list of the #include files specified in the source. This text file shares the same name as the source file but with the extension, .INC. The files are stored in the path specified by the [TEXTPATH](#) environment variable. The generated file may be used in make files.

Example

-Li

If the source file is 'c:\myFiles\b.c':

```
/* c:\myFiles\b.c */
#include <string.h>
```

Then the generated file is

```
c:\myFiles\b.c :    \
c:\Metrowerks\lib\targetc\include\string.h \
c:\Metrowerks\lib\targetc\include\libdefs.h \
c:\Metrowerks\lib\targetc\include\hedef.h \
c:\Metrowerks\lib\targetc\include\stddef.h \
c:\Metrowerks\lib\targetc\include\stdtypes.h
```

See also

[Option -Lm](#)

-Lic: License Information

Group

VARIOUS

Scope

None

Syntax

"-Lic"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The -Lic option prints the current license information (e.g. if it is a demo version or a full version). This information is also displayed in the about box.

Example

-Lic

See also

[Option -LicA](#)

[Option -LicBorrow](#)

[Option -LicWait](#)

-LicA: License Information about every Feature in Directory

Group

VARIOUS

Scope

None

Syntax

"-LicA"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The -LicA option prints the license information (e.g. if the tool or feature is a demo version or a full version) of every tool or .dll in the directory where the executable is located. This will take some time as every file in the directory is analyzed.

Example

-LicA

See also

[Option -Lic](#)

[Option -LicBorrow](#)

Using the Compiler

Compiler Options

[Option -LicWait](#)

-LicBorrow: Borrow License Feature

Group

HOST

Scope

None

Syntax

"-LicBorrow"<feature>[","<version>]:"<Date>

Arguments

<feature>: the feature name to be borrowed (e.g. HI100100).

<version>: optional version of the feature to be borrowed (e.g. 3.000).

<date>: date with optional time until when the feature shall be borrowed (e.g. 15-Mar-2004:18:35).

Default

None

Defines

None

Pragmas

None

Description

This option allows to borrow a license feature until a given date/time. Borrowing allows you to use a floating license even if disconnected from the floating license server.

You need to specify the feature name and the date until you want to borrow the feature. If the feature you want to borrow is a feature belonging to the tool where you use this option, then you do not need to specify the version of the feature (because the tool knows the version). However, if you want to borrow any feature, you need to specify as well the feature version of it.

You can check the status of currently borrowed features in the tool about box.

NOTE	You only can borrow features, if you have a floating license and if your floating license is enabled for borrowing. See as well the provided FLEXlm documentation about details on borrowing.
-------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Example

-LicBorrowHI100100;3.000:12-Mar-2004:18:25

See also

[Option -LicA](#)

[Option -Lic](#)

[Option -LicWait](#)

-LicWait: Wait until floating License is available from floating License Server

Group

HOST

Scope

None

Syntax

"-LicWait"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

By default, if a license is not available from the floating license server, then the application will immediately return. With -LicWait set, the application will wait (blocking) until a license is available from the floating license server.

Example

-LicWait

See also

[Option -Lic](#)

Using the Compiler

Compiler Options

[Option -LicA](#)

[Option -LicBorrow](#)

-L1: Statistics About Each Function

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-L1" ["=" <file Name>]

Arguments

<file Name>; file to be used for the output

Default

The default output file name is `logfile.txt`

Defines

None

Pragmas

None

Description

The `-L1` option causes the Compiler to append statistical information about the compilation session to the specified file. Compiler options, code size (in bytes), stack usage (in bytes) and compilation time (in seconds) are given for each procedure of the compiled file. The information is appended to the specified file name (or the file 'make.txt', if no argument given). If the [TEXTPATH](#) environment variable is set, the file is stored into the path specified by the environment variable. Otherwise it is stored in the current directory.

Example

```
-Ll=mylog.txt

/* foo.c */
int Func1(int b) {
    int a = b+3;
    return a+2;
}
void Func2(void) {
```

Appends the following two lines into mylog.txt:

```
foo.c Func1 -Ll=mylog.txt    11  4  0.055000
foo.c Func2 -Ll=mylog.txt      1  0  0.001000
```

See also

None

-Lm: List of Included Files in make Format

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-Lm" ["=" <file Name>]

Arguments

<file Name>; file to be used for the output

Default

The default file name is `Make.txt`

Defines

None

Pragmas

None

Description

The `-Lm` option causes the Compiler to generate a text file which contains a list of the `#include` files specified in the source. The generated list is in a *make* format. The `-Lm` option is useful when creating make files. The output from several source files may be copied and grouped into one make file. The generated list is in the make format. The file names does not include the path. After each entry, an empty line is added. The information is appended to the specified file name (or the file 'make.txt', if no argument given). If the [TEXTPATH](#) environment variable is set, the file is stored into the path specified by the environment variable. Otherwise it is stored in the current directory.

Example

```
COMPOTIONS=-Lm=mymake.txt
```

Compiling the following sources 'foo.c' and 'second.c':

```
/* foo.c */
#include <stddef.h>
#include "myheader.h"
...
/* second.c */
#include "inc.h"
#include "header.h"
...
```

This adds the following entries in the 'mymake.txt':

```
foo.o :      foo.c stddef.h myheader.h
seconde.o : second.c inc.h header.h
```

See also

[Option -Li](#)
[Option -Lo](#)
Make Utility

-LmCfg: Configuration of List of Included Files in make Format

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-LmCfg" ["=" {"i" | "l" | "m" | "o" | "u"}]

Arguments

- i: Write path of included files
- l: Use line continuation
- m: Write path of main file
- o: Write path of object file
- u: Update information

Default

None

Defines

None

Pragmas

None

Description

This option is used when configuring the [option -Lm](#). This option is operative only if the -Lm option is also used. The -Lm option produces the ‘dependency’ information for a make file. Each dependency information grouping is structured as shown:

```
<main object file>: <main source file> {<included file>}
```

Example

If you compile a file named `b.c`, which includes ‘`stdio.h`’, the output of `-Lm` may be:

```
b.o: b.c stdio.h stddef.h stdarg.h string.h
```

The suboption, ‘l’, uses line continuation for each single entry in the dependency list. This improves readability as shown:

```
b.o: \
  b.c \
  stdio.h \
  stddef.h \
  stdarg.h \
  string.h
```

With the suboption ‘m’, the full path of the main file is written. The main file is the actual compilation unit (file to be compiled). This is necessary if there are files with the same name in different directories:

```
b.o: c:\test\b.c stdio.h stddef.h stdarg.h string.h
```

The suboption ‘o’ has the same effect as ‘m’, but writes the full name of the target object file:

```
c:\test\obj\b.o: b.c stdio.h stddef.h stdarg.h string.h
```

The suboption ‘i’ writes the full path of all included files in the dependency list:

```
b.o: b.c c:\Metrowerks\lib\include\stdio.h
c:\Metrowerks\lib\include\stddef.h
c:\Metrowerks\lib\include\stdarg.h
c:\Metrowerks\lib\include\
c:\Metrowerks\lib\include\string.h
```

The suboption ‘u’ updates the information in the output file. If the file does not exist, the file is created. If the file exists and the current information is not yet in the file, the information is appended to the file. If the information is already present, it is updated. This allows you to specify this suboption for each compilation ensuring that the make dependency file is always up to date.

Example

```
COMPOTIONS=-LmCfg=u
```

See also

[Option -Li](#)
[Option -Lo](#)

[Option -Lm](#)
Make Utility

-Lo: Object File List

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-Lo" ["=" <file Name>]

Arguments

<file Name>: file to be used for the output

Default

The default file name is objlist.txt

Defines

None

Pragmas

None

Description

The -Lo option causes the Compiler to append the object file name to the list in the specified file. The information is appended to the specified file name (or the file 'make.txt', if no argument given). If [TEXTPATH](#) is set, the file is stored into the path specified by the environment variable. Otherwise, it is stored in the current directory.

Example

-Lo

See also

[Option -Li](#)
[Option -Lm](#)

-Lp: Preprocessor Output

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-Lp" ["=" <filename>]

Arguments

<filename>: The name of the file to be generated.

It may contain special modifiers (see [Using Special Modifiers](#))

Default

No file created

Defines

None

Pragmas

None

Description

The -Lp option causes the Compiler to generate a text file which contains the preprocessor's output. If no file name is specified, the text file shares the same name as the source file but with the extension, .PRE (%on.pre). The TEXTPATH environment variable is used to store the preprocessor file.

The resultant file is a form of the source file. All preprocessor commands (i.e. #include, #define, #ifdef, etc.) have been resolved. Only source code is listed with line numbers.

Example

-Lp

See also

[Option -LpX](#)
[Option -LpCfg](#)

-LpCfg: Preprocessor Output configuration

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-LpCfg" ["=" {"c" | "f" | "l" | "s"}]

Arguments

- "c": Do not emit line comments
- "e": Emit empty lines
- "f": Filenames with path
- "l": Emit #line directives in preprocessor output
- "m": Do not emit file names
- "s": Maintain spaces

Default

If `-LpCfg` is specified, all suboptions (arguments) are enabled

Defines

None

Pragmas

None

Description

The `-LpCfg` option specifies how source file and -line information is formatted in the preprocessor output. Switching `-LpCfg` off means that the output is formatted like in former compiler versions. The effects of the arguments are listed in [Table 1.19](#).

Table 1.19 Effects of Source and Line Information Format Control Arguments

Argument	on	off
"c"	#line 1 #line 10	/* 1 */ /* 2 */ /* 10 */
"e"	int j; int i;	int j; int i;
"f"	c:\metrowerks\include\stdlib.h	stdlib.h
"I"	#line 1 "stdlib.h"	***** FILE 'stdlib.h' */
"m"		***** FILE 'stdlib.h' */
"s"	/* 1 */ int f(void) { /* 2 */ return 1; /* 3 */ }	/* 1 */ int f (void) { /* 2 */ return 1 ; /* 3 */ }
all	#line 1 "c:\metrowerks\include\stdlib.h" #line 10	***** FILE 'stdlib.h' */ /* 1 */ /* 2 */ /* 10 */

Example

```
-Lpcfg
-Lpcfg=lfs
```

See also

[Option -Lp](#)

-LpX: Stop After Preprocessor

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-LpX"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Without this option, the compiler always translates the preprocessor output as C/C++ code. To do only preprocessing, use this option together with the -Lp option. No object file is generated.

Example

-LpX

See also

[Option -Lp](#)

-N: Display Notify Box

PC

Group

MESSAGE

Scope

Function

Syntax

"-N"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Makes the Compiler display an alert box if there was an error during compilation. This is useful when running a make file (please see *Make Utility*) since the Compiler waits for you to acknowledge the message, thus suspending make file processing. The 'N' stands for "Notify".

This feature is useful for halting and aborting a build using the Make Utility.

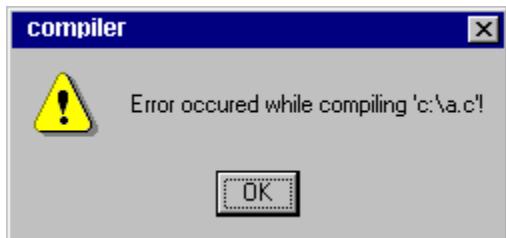
Example

-N

If during compilation an error occurs, a dialog box similar to the following one will be opened:

Using the Compiler

Compiler Options



See also

None

-NoBeep: No Beep in Case of an Error

Group

MESSAGE

Scope

Function

Syntax

"-NoBeep"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

There is a ‘beep’ notification at the end of processing if an error was generated. To implement a silent error, this ‘beep’ may be switched off using this option.

Example

-NoBeep

See also

None

-NoDebugInfo: Do Not Generate Debug Information

Group

Output

Scope

None

Syntax

`"-NoDebugInfo"`

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

The compiler generates debug information by default. When this option is used, the compiler does not generate debug information.

NOTE	To generate an application without debug information in ELF, the linker provides an option to strip the debug information. By calling the linker twice, you can generate two versions of the application: one with and one without debug information. This compiler option has to be used only if object files or libraries are to be distributed without debug info.
-------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

NOTE	This option does not affect the generated code. Only the debug information is excluded.
-------------	-----------------------------------------------------------------------------------------

Example

`-NoDebugInfo`

See also

[Option -F](#)

[Option -NoPath](#)

-NoEnv: Do Not Use Environment

Group

Startup. This option can not be specified interactively.

Scope

None

Syntax

"-NoEnv"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option can only be specified at the command line while starting the application. It can not be specified in any other way, including via the default.env file, the command line, or processes.

When this option is given, the application does not use any environment (default.env, project.ini or tips file) data.

Example

```
compiler.exe -NoEnv
```

Use the compiler executable name instead of “compiler”.

See also

[Section Configuration File](#)

-NoPath: Strip Path Info

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-NoPath"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

With this option set, it is possible to avoid any path information in object files. This is useful if you want to move object files to another file location, or to hide your path structure.

Example

-NoPath

See also

[Option -NoDebugInfo](#)

-Oa: Alias Analysis Options

SICG

Group

OPTIMIZATIONS

Scope

Function

Syntax

"-Oa" ("addr"|"ANSI"|"type"|"none")

Arguments

- "addr": All objects in same address area may overlap (safe mode, default)
- "ANSI": use ANSI99 rules
- "type": only objects in same address area with same type may overlap
- "none": assume no objects do overlap

Default

"addr"

Defines

None

Pragmas

None

Description

These 4 different options allow the programmer to control the alias behavior of the compiler. The option -oaaddr is the default because it is safe for all C programs. Use the option -oanssi if the source code follows the ANSI C99 alias rules. If objects with different types do never overlap in your program, use option -oatype. If your program doesn't have aliases at all, use option -oanone (or the ICG option -ona which is supported for compatibility reason).

Examples

-oAANSI

See also

none

-O (-Os, -Ot): Main Optimization Target

Group

OPTIMIZATIONS

Scope

Function

Syntax

"-O" ("s" | "t")

Arguments

"s": Optimization for code size (default)

"t": Optimization for execution speed

Default

-Os

Defines

__OPTIMIZE_FOR_SIZE__

__OPTIMIZE_FOR_TIME__

Pragmas

None

Description

There are various points where the Compiler has to choose between two possibilities: it can either generate fast, but large code, or small but slower code.

The Compiler generally optimizes on code size. It often has to decide between a runtime routine or an expanded code. The programmer can decide whether to choose between the slower and shorter or the faster and longer code sequence by setting a command line switch.

The -Os option directs the Compiler to optimize the code for smaller code size. The Compiler trades faster-larger code for slower-smaller code.

The -Ot option directs the Compiler to optimize the code for faster execution time. The Compiler will “trade” slower-smaller code for faster-larger code.

NOTE This option only affects some special code sequences. This option has to be set together with other optimization options (e.g. register optimization) to get best results.

Example

-Os

-ObjN: Object File Name Specification

Group

OUTPUT

Scope

Compilation Unit

Syntax

"-ObjN=<file>.

Arguments

<file>: Object file name

Default

-ObjN=%(OBJPATH)\%n.o

Defines

None

Pragmas

None

Description

The object file has the same name as the processed source file, but with the extension ".o". This option allows a flexible way to define the object file name. It may contain special modifiers (see [Using Special Modifiers](#)). If <file> in the option contains a path (absolute or relative), the OBJPATH environment variable is ignored.

Example

-ObjN=a.out

The resulting object file is "a.out". If the OBJPATH environment variable is set to "\src\obj", the object file is "\src\obj\a.out".

fibo.c -ObjN=%n.obj

The resulting object file is “fibo.obj”.

```
myfile.c -ObjN=..\objects\_%n.obj
```

The object file is named relative to the current directory to
“..\objects_myfile.obj”. The OBJPATH environment variable is ignored
as the <file> contains a path.

See also

[Environment variable OBJPATH](#)

-Oc: Common Subexpression Elimination (CSE)

Group

OPTIMIZATIONS

Scope

Function

Syntax

"-Oc"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Performs common subexpression elimination (CSE). The code for common subexpressions and assignments is generated only once. The result is reused. Depending on available registers, a common subexpression may produce more code due to many spills.

NOTE

When the CSE is switched on, changes of variables by aliases may generate incorrect optimizations.

Example

-Oc

Example where CSE may produce wrong results

```
void main (void) {
    int x;
    int *p;
    x = 7; /* here the value of x is set to 7 */
    p = &x;
    *p = 6; /* here x is set to 6 by the alias *p */
    /* here x is assumed to be equal to 7 and
       Error is called */
    if(x != 6) Error();
}
```

NOTE This error does not occur if x is declared as volatile.

See also

None

-Od: Disable mid-level Optimizations

SICG

Group

OPTIMIZATIONS

Scope

Function

Syntax

"-Od" ["=" <option Char> {<option Char>}]

Arguments

<option Char> is one of the following:

- "a": Disable mid level copy propagation
- "b": Disable mid level constant propagation
- "c": Disable mid level common subexpression elimination (CSE)
- "d": Disable mid level removing dead assignments
- "e": Disable mid level instruction combination
- "f": Disable mid level code motion
- "g": Disable mid level loop induction variable elimination

Default

None

Defines

None

Pragmas

None

Description

The backend of this compiler is based on the new 2nd generation intermediate code generator (SICG). All intermediate language and processor independent

optimizations (cf. NULLSTONE) are performed by the SICG optimizer using the powerful static single assignment form (SSA form). The optimizations are switched off using -od. Currently four optimizations are implemented. More will follow soon.

Examples

- Od
disables all mid-level optimizations
- Od=d
disables only removing dead assignments
- Od=cd
disables removing dead assignments and CSE

See also

None

-Odb: Disable mid-level Branch Optimizations

SICG

Group

OPTIMIZATIONS

Scope

Function

Syntax

"-Odb" ["=" <option Char> {<option Char>}]

Arguments

<option Char> is one of the following:

"a": Disable mid level label rearranging

"b": Disable mid level branch tail merging

"c": Disable mid level loop hoisting

Default

None

Defines

None

Pragmas

None

Description

This option disables branch optimizations on the SSA form based on control flows.

Label rearranging sorts all labels of the control flow to generate a minimum amount of branches.

Branch tail merging places common code into joining labels, as shown:

```
void foo(void) {void foo(void) {  
if(cond) {if(cond) {
```

Using the Compiler

Compiler Options

```
.....  
a = 0;} else {  
} else {...  
...}  
a = 0;a = 0;  
}}  
}
```

Examples

-Odb
disables all mid-level branch optimizations

-Odb=b
disables only branch tail merging

See also

None

-OdocF: Dynamic Option Configuration for Functions

Group

OPTIMIZATIONS

Scope

Function

Syntax

`"-OdocF=" "<option>"`

Arguments

`<options>`: Set of options, separated by ‘|’> to be evaluated for each single function.

Default

None

Defines

None

Pragmas

None

Description

Normally, you must set a specific set of Compiler switches for each compilation unit (file to be compiled). For some files, a specific set of options may decrease the code size, but for other files, the same set of Compiler options may produce more code depending on the sources.

Some optimizations may reduce the code size for some functions, but may increase the code size for other functions in the same compilation unit. Normally it is impossible to vary options over different functions, or to find the best combination of options.

This option solves this problem by allowing the Compiler to choose from a set of options to reach the smallest code size for every function. Without this feature,

you must set some Compiler switches, which are fixed, over the whole compilation unit. With this feature, the Compiler is free to find the best option combination from a user-defined set for every function.

Standard merging rules applies also for this new option, e.g.

`-Or -OdocF= "-Ocu | -Cu"`

is the same as

`-OrDOCF= "-Ouc | -Cu"`

The Compiler attempts to find the best option combination (of those specified) and evaluates all possible combinations of all specified sets, e.g. for the following option:

`-W2 -OdocF= "-Or | -Cni -Cu | -Oc"`

The code sizes for following option combinations are evaluated:

1. `-W2`
2. `-W2 -Or`
3. `-W2 -Cni -Cu`
4. `-W2 -Or -Cni -Cu`
5. `-W2 -Oc`
6. `-W2 -Or -Oc`
7. `-W2 -Cni -Cu -Oc`
8. `-W2 -Or -Cni -Cu -Oc`

Thus, if the more sets are specified, the longer the Compiler has to evaluate all combinations, e.g. for 5 sets 32 evaluations.

NOTE	No options with scope Application or Compilation Unit (as memory model, float/double format/object file format) or options for the whole compilation unit (like inlining or macro definition) should be specified in this option. The generated functions may be incompatible for linking and executing.
-------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Limitations:

- The maximum set of options set is limited to five, e.g. `'-OdocF="-Or -Ou|-Cni|-Cu|-Oic2|-W2 -Ob"`
- The maximum length of the option is 64 characters.
- The feature is available only for functions and options compatible with functions. Future extensions will also provide this option for compilation units.

Example

`-Odocf=" -Or | -Cni"`

See also

None

-Oi: Inlining

Group

OPTIMIZATIONS

Scope

Compilation unit

Syntax

"-Oi" ["=" ("c"<code Size> | "OFF")]

Arguments

<code Size>: Limit for inlining in code size

OFF: switching off inlining

Default

None. If no <code Size> is specified, the compiler uses a default code size of 40 bytes

Defines

None

Pragmas

```
#pragma INLINE
```

Description

This option enables inline expansion. If there is a `#pragma INLINE` before a function definition, or the C++ keyword “`inline`” is used, all calls of this function are replaced by the code of this function, if possible.

Using the option `-Oi=c0` switches off inlining. Functions marked with the `#pragma INLINE` are still inlined. To disable inlining, use the `-Oi=OFF` option.

Example

```
-Oi

#pragma INLINE
static void f(int i) {
    /* all calls of function f() are inlined */
    /* ... */
}

inline static void g(int i) { /* C++ only! */
    /* all calls of function g() are inlined */
    /* ... */
}
```

The option extension [=c<n>] signifies that all functions with a size smaller than <n> are inlined. For example, compiling with the option `-oi=c100` enables inline expansion for all functions with a size smaller than 100 bytes.

Restrictions

The following functions are not inlined:

- functions with default arguments
- functions with labels inside
- functions with an open parameter list (“`void f(int i,...);`”)
- functions with inline assembly statements
- functions using local static objects

See also

None

-Oilib: Inline Library Functions

Group

OPTIMIZATIONS

Scope

Function

Syntax

"-Oilib" ["=" <arguments>]

Arguments

<arguments> are one or multiple of following suboptions:

- a: inline calls to function “strcpy”
- b: inline calls to function “strlen”
- d: inline calls to function “fabs/fabsf”
- e: inline calls to function “memset”
- f: inline calls to function “memcpy”
- g: replace shifts left of 1 by array lookup

Default

None

Defines

None

Pragmas

None

Description

This option enables the compiler, when inlining specific library functions, to reduce execution time. The Compiler frequently uses small functions such as strcpy, strcmp, and so forth. The following functions are inlined:

- strcpy (only available for ICG based back ends)
- strlen (e.g. strlen(“abc”))

- fabs/fabs (e.g. ‘f = fabs(f);’)
- memset

memset() is inlined only if:

- the result is not used
- memset is used to zero out
- the size for the zero out is in the range 1 – 0xff
- the ANSI library header file <string.h> is included

An example for this is ‘(void)memset(&buf, 0, 50);’ In this case, the call to memset() is replaced with a call to ‘_memset_clear_8bitCount’ present in the ANSI library (string.c)

memcpy() is inlined only if:

- the result is not used
- the size for the copy out is in the range 0 – 0xff
- the ANSI library header file <string.h> is included

An example for this is ‘(void)memcpy(&buf, &buf2, 30);’

In this case the call to memcpy() is replaced with a call to ‘_memcpy_8bitCount’ present in the ANSI library (string.c)

(char)1 << val is replaced by _PowOfTwo_8[val] if _PowOfTwo_8 is known at compile time. Similarly, for 16 bit and for 32 bit shifts, the arrays

_PowOfTwo_16 and _PowOfTwo_32 are used. These constant arrays contain the values 1,2,4,8... . They are declared in hidef.h. This optimization is performed only when optimizing for time.

-Oilib without arguments: inline calls to all supported library functions.

Example

Compiling function f below with option -Oilib=a

```
void f(void) {
    char *s = strcpy(s, ct);
}
```

is translated similar to the following function:

```
void g(void) {
    s2 = s;
    while(*s2++ = *ct++);
}
```

See also

[Option -Oi](#)
[Message C5920](#)

-OnBRA: Disable JAL to BRA Optimization

XGATE

Group

OPTIMIZATIONS

Scope

Function

Syntax

"-OnBRA"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

If the call distance to a subroutine defined in the same compilation unit is in the range of [-512 – 512], the Compiler replaces a JAL instruction by a BRA instruction to reduce code size. Disable this optimization by specifying the option -OnBRA option if your linker places code between caller and callee.

Example1: Branch to Subroutine

```
int f(void) {  
    return 1;  
}  
int g(void) {  
    return f();  
}
```

with -OnBRA:

```
  X_f:  
F201 LDL         R2, #1  
06F6 JAL         R6  
  X_g:  
7E1E STW         R6, (R0,-R7)  
    5: return f();  
F600 LDL         R6, #%XGATE_8(_X_f)  
AE00 ORH         R6, #%XGATE_8_H(_X_f)  
06F6 JAL         R6  
6E1D LDW         R6, (R0,R7+)  
06F6 JAL         R6
```

without -OnBRA:

```
  X_f:  
F201 LDL         R2, #1  
    3: }  
06F6 JAL         R6  
  X_g:  
7E1E STW         R6, (R0,-R7)  
    5: return f();  
06FA TFR         R6, PC  
3C00 BRA           X_f  
6E1D LDW         R6, (R0,R7+)  
06F6 JAL         R6
```

Example2: Conditional Branch to Subroutine

```
int a;  
int f(void) {  
    return 1;  
}  
void g(void) {  
    if (a != 0) {  
        (void)f();  
    }  
}
```

with -OnBRA:

```
  X_f:  
F201 LDL         R2, #1  
06F6 JAL         R6  
  X_g:
```

```

7E1E STW      R6, (R0,-R7)
if (a != 0) {
F200 LDL      R2, #%XGATE_8(a)
AA00 ORH      R2, #%XGATE_8_H(a)
4A40 LDW      R2, (R2,#0)
1840 TST      R2
2603 BEQ      *+8      ;abs = 0x00000016
(void)f();
F600 LDL      R6, #%XGATE_8(_X_f)
AE00 ORH      R6, #%XGATE_8_H(_X_f)
06F6 JAL      R6

6E1D LDW      R6, (R0,R7+)
06F6 JAL      R6

```

without -OnBRA:

```

_X_f:
F201 LDL      R2, #1
06F6 JAL      R6
_X_g:
7E1E STW      R6, (R0,-R7)
6: if (a != 0) {
F200 LDL      R2, #%XGATE_8(a)
AA00 ORH      R2, #%XGATE_8_H(a)
4A40 LDW      R2, (R2,#0)
1840 TST      R2
06FA TFR      R6, PC
2400 BNE      _X_f
9: }
6E1D LDW      R6, (R0,R7+)
06F6 JAL      R6

```

Example

-OnBRA

See also

none

-OnCopyDown: Do Generate Copy Down Information for Zero Values

Group

OPTIMIZATIONS

Scope

Compilation unit

Syntax

"-OnCopyDown"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

With usual startup code, all global variables are first set to 0 (zero out). If the definition contained an initialization value, this initialization value is copied to the variable (copy down). Because of this, it is not necessary to copy zero values unless the usual startup code is modified. If a modified startup code contains a copy down but not a zero out, use this option to prevent the compiler from removing the initialization.

NOTE

The case of a copy down without a zero out is normally not used. Because the copy down needs much more space than the zero out, it usually contains copy down and zero out, zero out alone, or none of them.

In the HIWARE format, the object file format permits the Compiler to remove single assignments in a structure or array initialization. In the ELF format, it is optimized only if the whole array or structure is initialized with 0.

NOTE

This option controls the optimizations done in the compiler. However, the linker itself might further optimize the copy down or the zero out.

Example

```
int i=0;  
int arr[10]={1,0,0,0,0,0,0,0,0,0};
```

If this option is present, no copy down is generated for `i`. For the array `arr`, the initialization with 0 can only be optimized in the HIWARE format. In ELF it is not possible to separate them from the initialization with 1.

See also

None

-OnCstVar: Disable CONST Variable by Constant Replacement

Group

OPTIMIZATIONS

Scope

Compilation Unit

Syntax

"-OnCstVar"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option provides you with a way to switch OFF the replacement of CONST variable by the constant value.

Example

```
const int MyConst = 5;
int i;
void foo(void) {
    i = MyConst;
}
```

If the option -OnStVar is not set, the compiler replaces each occurrence of 'MyConst' with its constant value 5; that is 'i = MyConst' will be transformed

into ‘`i = 5;`’. The Memory/ROM needed for the ‘`MyConst`’ constant variable is optimized as well. With the `-OnCstVar` option set, this optimization is avoided. This is logical only if you want to have unoptimized code.

See also

None

-OnPMNC: Disable Code Generation for NULL Pointer to Member Check

Group

OPTIMIZATIONS

Scope

Compilation Unit

Syntax

"-OnPMNC"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Before assigning a pointer to member in C++, you must ensure that the pointer to member is not NULL in order to generate correct and safe code. In embedded systems development, the problem is to generate the denser code while avoiding overhead whenever possible (this NULL check code is a good example). If you can ensure this pointer to member will never be NULL, then this NULL check is useless. This option enables you to switch off the code generation for the NULL check.

Example

-OnPMNC

See also

None

-Ont: Disable Tree Optimizer

Group

OPTIMIZATIONS

Scope

Function

Syntax

```
"-Ont" [= {"%" | "&" | "*" | "+" | "-" | "/" | "0" | "1" | "7" | "8" | "9" | "?" | "^" |
    "a" | "b" | "c" | "d" | "e" | "f" | "h" | "i" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" |
    "u" | "v" | "w" | "|" | "~"}]
```

Arguments

- "%": Disable mod optimization
- "&": Disable bit and optimization
- "*": Disable mul optimization
- "+": Disable plus optimization
- "-": Disable minus optimization
- "/": Disable div optimization
- "0": Disable and optimization
- "1": Disable or optimization
- "7": Disable extend optimization
- "8": Disable switch optimization
- "9": Disable assign optimization
- "?": Disable test optimization
- "^": Disable xor optimization
- "a": Disable statement optimization
- "b": Disable constant folding optimization
- "c": Disable compare optimization
- "d": Disable binary operation optimization
- "e": Disable constant swap optimization
- "f": Disable condition optimization
- "h": Disable unary minus optimization
- "i": Disable address optimization
- "j": Disable transformations for inlining
- "l": Disable label optimization
- "m": Disable left shift optimization

"n": Disable right shift optimization
"o": Disable cast optimization
"p": Disable cut optimization
"q": Disable 16-32 compare optimization
"r": Disable 16-32 relative optimization
"s": Disable indirect optimization
"t": Disable for optimization
"u": Disable while optimization
"v": Disable do optimization
"w": Disable if optimization
"~":" Disable bit or optimization
"~":" Disable bit neg optimization

Default

If -Ont is specified, all optimizations are disabled

Defines

None

Pragmas

None

Description

The Compiler contains a special optimizer which optimizes the internal tree data structure. This tree data structure holds the semantic of the program and represents the parsed statements and expressions.

This option disables the tree optimizer. This may be useful for debugging, and to force the Compiler to produce ‘straight forward’ code. Note that the optimizations below are just examples for the classes of optimizations.

If this option is set, the Compiler will not perform the following optimizations:

-Ont=~

Disable optimization of ‘ $\sim i$ ’ into ‘ i ’

-Ont=|

Disable optimization of ‘ $i|0xffff$ ’ into ‘ $0xffff$ ’

-Ont=w

Disable optimization of ‘if (1) i = 0;’ into ‘i = 0;’

-Ont=v

Disable optimization of ‘do ... while(0) into ‘...’

-Ont=u

Disable optimization of ‘while(1) ...;’ into ‘...,’

-Ont=t

Disable optimization of ‘for(;;) ...’ into ‘while(1) ...’

-Ont=s

Disable optimization of ‘*&i’ into ‘i’

-Ont=r

Disable optimization of ‘L<=4’ into 16bit compares if 16bit compares are better

-Ont=q

Reduction of long compares into int compares if int compares are better: (-Ont=q to disable it)

`if (uL == 0)`

will be optimized into

`if ((int)(uL>>16) == 0 && (int)uL == 0)`

-Ont=p

Disable optimization of ‘(char)(long)i’ into ‘(char)i’

-Ont=o

Disable optimization of ‘(short)(int)L’ into ‘(short)L’ if short and int have the same size

-Ont=n, -Ont=m:

Optimization of shift optimizations (<<, >>, -Ont=n or -Ont=m to disable it):
Reduction of shift counts to unsigned char:

`uL = uL1 >> uL2;`

will be optimized into

```
uL = uL1 >> (unsigned char)uL2;
```

Optimization of zero shift counts:

```
uL = uL1 >> 0;
```

will be optimized into

```
uL = uL1;
```

Optimization of shift counts greater than the object to be shifted:

```
uL = uL1 >> 40;
```

will be optimized into

```
uL = 0L;
```

Strength reduction for operations followed by a cut operation:

```
ch = uL1 * uL2;
```

will be optimized into

```
ch = (char)uL1 * (char)uL2;
```

Replacing shift operations by load/store

```
i = uL >> 16;
```

will be optimized into

```
i = *(int *)(&uL);
```

Shift count reductions:

```
ch = uL >> 17;
```

will be optimized into

```
ch = (*((char *)(&uL)+1))>>1;
```

Optimization of shift combined with binary and:

```
ch = (uL >> 25) & 0x10;
```

will be optimized into

```
ch = (((*(char *)(&uL))>>1) & 0x10);
```

-Ont=I

Disable optimization removal of labels if not used

-Ont=i

Disable optimization of '&*p' into 'p'

-Ont=j

This optimization does transform the syntax tree into an equivalent form in which more inlining cases can be done. This option only has an effect when inlining is enabled.

-Ont=h

Disable optimization of '-(-i)' into 'i'

-Ont=f

Disable optimization of '(a==0)' into '(!a)'

-Ont=e

Disable optimization of '2*i' into 'i*2'

-Ont=d

Disable optimization of 'us & ui' into 'us & (unsigned short)ui'

-Ont=c

Disable optimization of 'if ((long)i)' into 'if (i)'

-Ont=b

Disable optimization of '3+7' into '10'

-Ont=a

Disable optimization of last statement in function if result is not used

-Ont=^

Disable optimization of 'i^0' into 'i'

-Ont=?

Disable optimization of 'i = (int)(cond ? L1:L2);' into
'i = cond ? (int)L1:(int)L2;'

-Ont=9

Disable optimization of ‘i=i;’

-Ont=8

Disable optimization of empty switch statement

-Ont=7

Disable optimization of ‘(long)(char)L’ into ‘L’

-Ont=1

Disable optimization of ‘a || 0’ into ‘a’

-Ont=0

Disable optimization of ‘a && 1’ into ‘a’

-Ont=/

Disable optimization of ‘a/1’ into ‘a’

-Ont=-

Disable optimization of ‘a-0’ into ‘a’

-Ont=+

Disable optimization of ‘a+0’ into ‘a’

-Ont=*

Disable optimization of ‘a*1’ into ‘a’

-Ont=&

Disable optimization of ‘a&0’ into ‘0’

-Ont=%

Disable optimization of ‘a%1’ into ‘0’

Example

`fibo.c -Ont`

See also

None

-Pe: Preprocessing Escape Sequences in Strings

Group

LANGUAGE

Scope

Compilation Unit

Syntax

"-Pe"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

If escape sequences are used in macros, they are handled in an include directive similar to the way they are handled in a printf instruction:

```
#define STRING "c:\myfile.h"
#include STRING
```

produces an error:

```
>> Illegal escape sequence
```

and used in:

```
printf(STRING);
```

produces a carriage return with line feed:

```
c:  
myfile
```

If the -Pe option is used, escape sequences are ignored in strings that contain a DOS drive letter ('a – 'z', 'A' – 'Z') followed by a colon ':' and a backslash '\'.

When the -Pe option is enabled, the Compiler handles strings in include directives differently from other strings. Escape sequences in include directive strings are not evaluated.

The following example:

```
#include "c:\names.h"
```

results in exactly the same include file name as in the source file ("c:\names.h"). If the file name appears in a macro, the Compiler does not distinguish between file name usage and normal string usage with escape sequence. This occurs because the macro STRING has to be the same for the include and the printf call, as shown:

```
#define STRING "c:\n.h"  
#include STRING /* means: "c:\n.h" */  
  
void main(void) {  
    printf(STRING); /* means: "c:", new line and ".h" */  
}
```

This option may be used to use macros for include files. This prevents escape sequence scanning in strings if the string starts with a DOS drive letter ('a – 'z', 'A' – 'Z') followed by a colon ':' and a backslash '\'. With the option set, the above example includes the 'c:\n.h' file and calls printf with "c:\n.h").

Example

-Pe

See also

None

-Pio: Include Files Only Once

Group

INPUT

Scope

Compilation Unit

Syntax

"-Pio"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Includes every header file only once. Whenever the compiler reaches an #include directive, it checks if this file to be included was already read. If so, the compiler ignores the #include directive. It is common practice to protect header files from multiple inclusion by conditional compilation, as shown:

```
/* Header file myfile.h */
#ifndef _MY_FILE_H_
#define _MY_FILE_H_

/* .... content .... */
#endif /* _MY_FILE_H_ */
```

When #ifndef and #define directives are issued, header file content is read only once even when the header file is included several times. This solves many

problems as C-language protocol does not allow you to define structures (such as enums or typedefs) more than once.

When all header files are protected this way, this option can safely accelerate the compilation.

This option must not be used when a header file must be included twice, e.g. the file contains macros which are set differently at the different inclusion times.

In those instances, [#pragma ONCE](#) is used to accelerate the inclusion of safe header files which do not contain macros of that nature.

Example

-Pio

See also

[#pragma ONCE](#)

-Prod: Specify Project File at Startup

Group

Startup - This option can not be specified interactively.

Scope

None

Syntax

"-Prod=" <file>

Arguments

<file>: name of a project or project directory

Default

None

Defines

None

Pragmas

None

Description

This option can only be specified at the command line while starting the application. It can not be specified in any other circumstances, including the default.env file, the command line or whatever.

When this option is given, the application opens the file as configuration file.

When the file name does only contain a directory, the default name project.ini is appended. When the loading fails, a message box appears.

Example

```
compiler.exe -prod=project.ini
```

Use the compiler executable name instead of "compiler".

Using the Compiler

Compiler Options

See also

[Section Configuration File](#)

-Qvtp: Qualifier for Virtual Table Pointers

C++

Group

CODE GENERATION

Scope

Application

Syntax

"-Qvtp" ("none" | "far" | "near" | "rom" | "uni" | "paged")

Arguments

None

Default

-Qvptnone

Defines

None

Pragmas

None

Description

Using a virtual function in C++ requires an additional pointer to virtual function tables. This pointer is not accessible and is generated by the compiler in every class object when virtual function tables are associated.

NOTE

It is useless to specify a qualifier which is not supported by the Back End (see Back End), e.g. using a 'far' qualifier if the Back End or CPU does not support any __far data accesses.

Example

-QvtpFar

This sets the qualifier for virtual table pointers to `_far` enabling the virtual tables to be placed into a `_FAR_SEG` segment (if the Back End/CPU supports `_FAR_SEG` segments).

See also

[C++ Front End](#)

-T: Flexible Type Management

Group

LANGUAGE.

Scope

Application

Syntax

"-T" <Type Format>

Arguments

<Type Format>: See below

Default

Depends on target, see Compiler Back End

Defines

To deal with different type sizes, one in the following define groups will be predefined by the Compiler:

```
__CHAR_IS_SIGNED__
__CHAR_IS_UNSIGNED__

__CHAR_IS_8BIT__
__CHAR_IS_16BIT__
__CHAR_IS_32BIT__
__CHAR_IS_64BIT__

__SHORT_IS_8BIT__
__SHORT_IS_16BIT__
__SHORT_IS_32BIT__
__SHORT_IS_64BIT__

__INT_IS_8BIT__
__INT_IS_16BIT__
__INT_IS_32BIT__
__INT_IS_64BIT__

__ENUM_IS_8BIT__
__ENUM_IS_16BIT__
```

```
__ENUM_IS_32BIT__
__ENUM_IS_64BIT__

__ENUM_IS_SIGNED__
__ENUM_IS_UNSIGNED__

__PLAIN_BITFIELD_IS_SIGNED__
__PLAIN_BITFIELD_IS_UNSIGNED__

__LONG_IS_8BIT__
__LONG_IS_16BIT__
__LONG_IS_32BIT__
__LONG_IS_64BIT__

__LONG_LONG_IS_8BIT__
__LONG_LONG_IS_16BIT__
__LONG_LONG_IS_32BIT__
__LONG_LONG_IS_64BIT__

__FLOAT_IS_IEEE32__
__FLOAT_IS_IEEE64__
__FLOAT_IS_DSP__

__DOUBLE_IS_IEEE32__
__DOUBLE_IS_IEEE64__
__DOUBLE_IS_DSP__

__LONG_DOUBLE_IS_IEEE32__
__LONG_DOUBLE_IS_IEEE64__
__LONG_DOUBLE_IS_DSP__

__LONG_LONG_DOUBLE_IS_IEEE32__
__LONG_LONG_DOUBLE_IS_IEEE64__
__LONG_LONG_DOUBLE_DSP__

__VTAB_DELTA_IS_8BIT__
__VTAB_DELTA_IS_16BIT__
__VTAB_DELTA_IS_32BIT__
__VTAB_DELTA_IS_64BIT__

__PTRMBR_OFFSET_IS_8BIT__
__PTRMBR_OFFSET_IS_16BIT__
__PTRMBR_OFFSET_IS_32BIT__
__PTRMBR_OFFSET_IS_64BIT__
```

Pragmas

None

Description

This option allows configurable type settings. The syntax of the option is: -
T{<type> <format>}

For <type> one of the keys listed in [Table 1.20](#) may be specified:

Table 1.20 Data Type Keys

Type	Key
char	'c'
short	's'
int	'i'
long	'L'
long long	'LL'
float	'f'
double	'd'
long double	'Ld'
long long double	'LLd'
enum	'e'
sign plain bitfield	'b'
virtual table delta size	'vtd'
pointer to member offset size	'pmo'

NOTE

Keys are not case sensitive, e.g. both 'f' and 'F' may be used for the type "float".

The sign of the type 'char,' or of the enumeration type, may be changed with a prefix placed before the key for the char key. See [Table 1.21](#).

Table 1.21 Keys for Signed And Unsigned Prefixes

Sign prefix	Key
signed	's'
unsigned	'u'

The sign of the type 'plain bitfield type' is changed with the options shown in [Table 1.22](#). Plain bitfield are bitfields defined/declared without an explicit signed/unsigned qualifier, e.g. 'int field:3'. Using this option, you can specify if the 'int' in the previous example is handled as 'signed int' or as 'unsigned int'. Note that this option may not be available on all targets. Also the default setting may vary. Refer to [Defines for Plain Bitfield Sign](#).

Table 1.22 Keys for Signed And Unsigned Bitfield Prefixes

Sign prefix	Key
plain signed bitfield	'bs'
plain unsigned bitfield	'bu'

For <format> one of the keys in [Table 1.23](#) can be specified.

Table 1.23 Data Format Specifier Keys

Format	Key
8 bit integral	'1'
16 bit integral	'2'
24 bit integral	'3'
32 bit integral	'4'
64 bit integral	'8'
IEEE32 floating	'2'
IEEE64 floating	'4'
DSP(32 bit)	'0'

Not all formats may be available for a target. See Back End for supported formats.

NOTE At least one type for each basic size (1, 2, 4 bytes) has to be available. It is illegal if no type of any sort is not set to at least a size of one. See Back End for default settings.

NOTE Enumeration types have the type ‘signed int’ by default for ANSI-C compliance.

The -Tpmo option allows you to change the pointer to a member offset value type. The default setting is 16 bits. The pointer to the member offset is used for C++ pointer to members only.

Example

-Tsc sets ‘char’ to ‘signed char’
and -Tuc sets ‘char’ to ‘unsigned char’

Example

-Tscl1s2i2L4LL4f2d4Ld4LLd4e2

denotes:

- signed char with 8 bit (sc1)
- short and int with 16 bit (s2i2)
- long, long long with 32 bit (L4LL4)
- float with IEEE32 (f2)
- double, long double and long long double with IEEE64 (d4Ld4LL4)
- enum with 16 bit (signed) (e2)

Restrictions:

For integrity and compliance to ANSI, the following two rules has to be true:

sizeof(char)	<= sizeof(short)
sizeof(short)	<= sizeof(int)
sizeof(int)	<= sizeof(long)
sizeof(long)	<= sizeof(long long)
sizeof(float)	<= sizeof(double)
sizeof(double)	<= sizeof(long double)
sizeof(long double)	<= sizeof(long long double)

NOTE It is illegal to set `char` to 16 bit and `int` to 8 bit.

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the Compiler are compiled with the standard type settings.

Also be careful if you change the type sizes for under or overflows, e.g. assigning a value too large to an object which is smaller now, as shown in the following example:

```
int i; /* -Til int has been set to 8 bit! */
i = 0x1234; /* i will set to 0x34! */
```

Examples:

Setting the size of `char` to 16 bit:

```
-Tc2
```

Setting the size of `char` to 16 bit and plain `char` is signed:

```
-Tsc2
```

Setting `char` to 8 bit and `unsigned`, `int` to 32 bit and `long long` to 32 bit

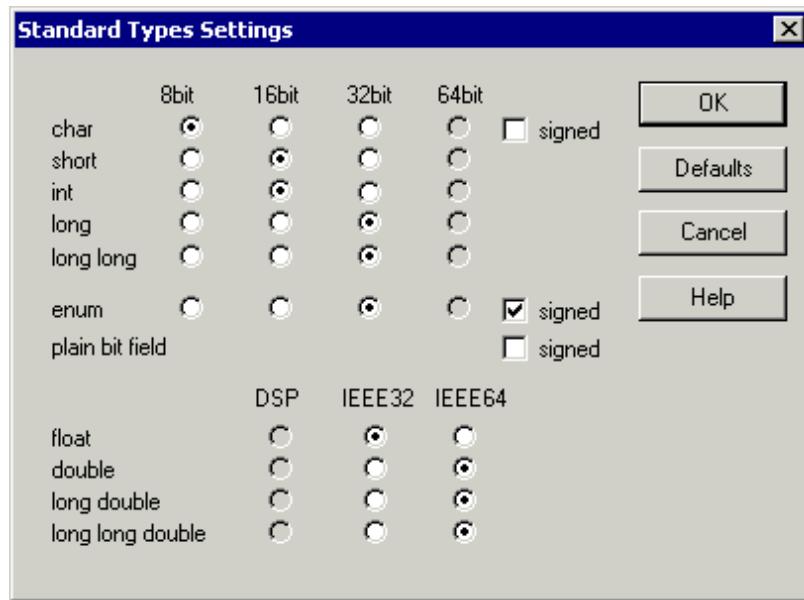
```
-Tuc1i4LL4
```

Setting `float` to IEEE32 and `double` to IEEE64:

```
-Tf2d4
```

The `-Tvtd` option allows you to change the delta value type inside virtual function tables (see also [C++ Front End](#)). The default setting is 16bit.

Another way to set this option is using the dialog in the Graphical User Interface:



See also

[Defines for Plain Bitfield Sign](#)

-V: Prints the Compiler Version

Group

VARIOUS

Scope

None

Syntax

"-V"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Prints the Compiler version of the internal subversion numbers of the parts the Compiler consists of and the current directory.

NOTE This option can determine the current directory of the Compiler.

Example

-V produces the following list:

```
Directory: \software\sources\c
ANSI-C Front End, V5.0.1, Date Jan 01 1997
```

Tree CSE Optimizer, V5.0.1, Date Jan 01 1997
Back End V5.0.1, Date Jan 01 1997

See also

None

PC

-View: Application Standard Occurrence

Group

HOST

Scope

Compilation Unit

Syntax

"-View" <kind>

Arguments

<kind> is one of:

“Window”: Application window has default window size

“Min”: Application window is minimized

“Max”: Application window is maximized

“Hidden”: Application window is not visible (only if arguments)

Default

Application started with arguments: Minimized

Application started without arguments: Window

Defines

None

Pragmas

None

Description

The application (e.g. linker, compiler, ...) is started as a normal window if no arguments are given. If the application is started with arguments (e.g. from the maker to compile/link a file), then the application runs minimized to allow batch processing.

You can specify the behavior of the application using this option. Using -ViewWindow, the application is visible with its normal window. Using -ViewMin the application is visible iconified (in the task bar). Using -ViewMax

the application is visible maximized (filling the hole screen). Using -ViewHidden the application processes arguments (e.g. files to be compiled/linked) completely invisible in the background (no window/icon in the task bar visible). However, if you are using the [-N](#) option, a dialog box is still possible.

Example

```
c:\Metrowerks\linker.exe -ViewHidden fibo.prm
```

See also

None

-WErrFile: Create "err.log" Error File

Group

MESSAGE

Scope

Compilation Unit

Syntax

"-WErrFile" ("On" | "Off")

Arguments

None

Default

err.log is created/deleted

Defines

None

Pragmas

None

Description

The error feedback to the tools that are called is done with a return code. In 16-bit window environments, this was not possible. In the error case, an "err.log" file, with the numbers of errors written into it, was used to signal an error. To state no error, the "err.log" file was deleted. Using UNIX or WIN32, there is now a return code available. The "err.log" file is no longer needed when only UNIX / WIN32 applications are involved.

NOTE: The error file must be created in order to signal any errors if you use a 16-bit maker with this tool.

Example

-WErrFileOn

The err.log file is created/deleted when the application is finished.

-WErrFileOff

The existing err.log file is not modified.

See also

[Option -WStdout](#)
[Option -WOutFile](#)

PC

-Wmsg8x3: Cut File Names in Microsoft Format to 8.3

Group

MESSAGE

Scope

Compilation Unit

Syntax

"-Wmsg8x3"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Some editors (e.g. early versions of WinEdit) expect the file name in the Microsoft message format (8.3 format). That means the file name can have, at most, eight characters with not more than a three-character extension. Longer file names are possible when you use Win95 or WinNT. This option truncates the file name to the 8.3 format.

Example

```
x:\mysourcefile.c(3): INFORMATION C2901: Unrolling loop
```

With the option -Wmsg8x3 set, the above message is:

```
x:\mysource.c(3): INFORMATION C2901: Unrolling loop
```

See also

[Option -WmsgFi](#)
[Option -WmsgFb](#)

-WmsgCE: RGB Color for Error Messages

Group

MESSAGE

Scope

Function

Syntax

"-WmsgCE" <RGB>

Arguments

<RGB>: 24bit RGB (red green blue) value

Default

-WmsgCE16711680 (rFF g00 b00, red)

Defines

None

Pragmas

None

Description

This option changes the error message color. The specified value must be an RGB (Red-Green-Blue) value, and must also be specified in decimal.

Example

-WmsgCE255 changes the error messages to blue

See also

None

-WmsgCF: RGB Color for Fatal Messages

Group

MESSAGE

Scope

Function

Syntax

"-WmsgCF" <RGB>

Arguments

<RGB>: 24bit RGB (red green blue) value

Default

-WmsgCF8388608 (r80 g00 b00, dark red)

Defines

None

Pragmas

None

Description

This option changes the color of a fatal message. The specified value must be an RGB (Red-Green-Blue) value, and must also be specified in decimal.

Example

-WmsgCF255 changes the fatal messages to blue

See also

None

-WmsgCI: RGB Color for Information Messages

Group

MESSAGE

Scope

Function

Syntax

"-WmsgCI" <RGB>

Arguments

<RGB>: 24bit RGB (red green blue) value

Default

-WmsgCI32768 (r00 g80 b00, green)

Defines

None

Pragmas

None

Description

This option changes the color of an information message. The specified value must be an RGB (Red-Green-Blue) value, and must also be specified in decimal.

Example

-WmsgCI255 changes the information messages to blue

See also

None

-WmsgCU: RGB Color for User Messages

Group

MESSAGE

Scope

Function

Syntax

"-WmsgCU" <RGB>

Arguments

<RGB>: 24bit RGB (red green blue) value

Default

-WmsgCU0 (r00 g00 b00, black)

Defines

None

Pragmas

None

Description

This option changes the color of a user message. The specified value must be an RGB (Red-Green-Blue) value, and must also be specified in decimal.

Example

-WmsgCU255 changes the user messages to blue

See also

None

-WmsgCW: RGB Color for Warning Messages

Group

MESSAGE

Scope

Function

Syntax

"-WmsgCW" <RGB>.

Arguments

<RGB>: 24bit RGB (red green blue) value

Default

-WmsgCW255 (r00 g00 bFF, blue)

Defines

None

Pragmas

None

Description

This option changes the color of a warning message. The specified value must be an RGB (Red-Green-Blue) value, and must also be specified in decimal.

Example

-WmsgCW0 changes the warning messages to black

See also

None

-WmsgFb (-WmsgFbi, -WmsgFbm): Set Message File Format for Batch Mode

Group

MESSAGE

Scope

Compilation Unit

Syntax

`"-WmsgFb" ["v" | "m"]`

Arguments

`"v"`: Verbose format

`"m"`: Microsoft format

Default

`-WmsgFbm`

Defines

None

Pragmas

None

Description

You can start the Compiler with additional arguments (e.g. files to be compiled together with Compiler options). If the Compiler has been started with arguments (e.g. from the Make Tool or with the `'%f'` argument from the CodeWright IDE), the Compiler compiles the files in a batch mode. No Compiler window is visible and the Compiler terminates after job completion.

If the Compiler is in batch mode, the Compiler messages are written to a file instead of to the screen. This file contains only the compiler messages (see examples below).

The Compiler uses a Microsoft message format to write the Compiler messages (errors, warnings, information messages) if the compiler is in batch mode.

This option changes the default format from the Microsoft format (only line information) to a more verbose error format with line, column, and source information.

NOTE Using the verbose message format may slow down the compilation because the compiler has to write more information into the message file.

Example

```
void foo(void) {  
    int i, j;  
    for(i=0;i<1;i++);  
}
```

The Compiler may produce the following file if it is running in batch mode (e.g. started from the Make tool):

```
X:\C.C(3): INFORMATION C2901: Unrolling loop  
X:\C.C(2): INFORMATION C5702: j: declared in function foo  
but not referenced
```

Setting the format to verbose, more information is stored in the file:

```
-WmsgFbv  
>> in "X:\C.C", line 3, col 2, pos 33  
    int i, j;  
  
    for(i=0;i<1;i++);  
  
    ^  
INFORMATION C2901: Unrolling loop  
>> in "X:\C.C", line 2, col 10, pos 28  
void foo(void) {  
  
    int i, j;  
  
    ^  
INFORMATION C5702: j: declared in function foo but not  
referenced
```

See also

[Environment variable ERRORFILE](#)

[Option -WmsgFi](#)

-WmsgFi (-WmsgFiv, -WmsgFim): Set Message Format for Interactive Mode

Group

MESSAGE

Scope

Compilation Unit

Syntax

`"-WmsgFi" ["v" | "m"]`

Arguments

"v": Verbose format

"m": Microsoft format

Default

`-WmsgFiv`

Defines

None

Pragmas

None

Description

The Compiler operates in the interactive mode (that is, a window is visible) if it is started without additional arguments (e.g. files to be compiled together with Compiler options).

The Compiler uses the verbose error file format to write the Compiler messages (errors, warnings, information messages).

This option changes the default format from the verbose format (with source, line and column information) to the Microsoft format (only line information).

NOTE

Using the Microsoft format may speed up the compilation because the compiler has to write less information to the screen.

Example

```
void foo(void) {
    int i, j;
    for(i=0;i<1;i++);
}
```

The Compiler may produce the following error output in the Compiler window if it is running in interactive mode:

```
Top: X:\C.C
Object File: X:\C.O

>> in "X:\C.C", line 3, col 2, pos 33
    int i, j;

    for(i=0;i<1;i++);

    ^
INFORMATION C2901: Unrolling loop
```

Setting the format to Microsoft, less information is displayed:

```
-WmsgFim

Top: X:\C.C
Object File: X:\C.O
X:\C.C(3): INFORMATION C2901: Unrolling loop
```

See also

[Environment variable ERRORFILE](#)

[Option -WmsgFb](#)

-WmsgFob: Message Format for Batch Mode

Group

MESSAGE

Scope

Function

Syntax

"-WmsgFob"<string>

Arguments

<string>: format string (see below).

Default

-WmsgFob"%""%f%e%"(%l): %K %d: %m\n"

Defines

None

Pragmas

None

Description

This option modifies the default message format in batch mode. The formats listed in [Table 1.24](#) are supported (assuming that the source file is x:\Metrowerks\mysourcefile.cpph):

Table 1.24 Message Format Specifiers

Format	Description	Example
%s	Source Extract	
%p	Path	x:\Metrowerks\
%f	Path and name	x:\Metrowerks\mysourcefile

Table 1.24 Message Format Specifiers

Format	Description	Example
%n	File name	mysourcefile
%e	Extension	.cpph
%N	File (8 chars)	mysource
%E	Extension (3 chars)	.cpp
%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space	
%'	A ' if the filename, the path or the extension contains a space	

Example

```
-WmsgFob"%f%e(%l): %k %d: %m\n"
```

Produces a message in the following format:

```
X:\C.C(3): information C2901: Unrolling loop
```

See also

[Environment variable ERRORFILE](#)

[Option -WmsgFb](#)

[Option -WmsgFi](#)

[Option -WmsgFonp](#)

[Option -WmsgFoi](#)

-WmsgFoi: Message Format for Interactive Mode

Group

MESSAGE

Scope

Function

Syntax

"-WmsgFoi"<string>

Arguments

<string>: format string (See below)

Default

-WmsgFoi"\n>> in "%f%e", line %l, col >>%c, pos %o\n%s\n%K %d: %m\n"

Defines

None

Pragmas

None

Description

This option modifies the default message format in interactive mode. The formats listed in [Table 1.25](#) are supported (assuming that the source file is x:\Metrowerks\mysourcefile.cpph):

Table 1.25 Message Format Specifiers

Format	Description	Example
%s	Source Extract	
%p	Path	x:\sources\
%f	Path and name	x:\sources\mysourcefile

Table 1.25 Message Format Specifiers

Format	Description	Example
%n	File name	mysourcefile
%e	Extension	.cpph
%N	File (8 chars)	mysource
%E	Extension (3 chars)	.cpp
%l	Line	3
%c	Column	47
%o	Pos	1234
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space.	
%'	A ' if the filename, the path or the extension contains a space	

Example

```
-WmsgFoi "%f%e(%l): %k %d: %m\n"
```

Produces a message in following format

```
X:\C.C(3): information C2901: Unrolling loop
```

See also

[Environment variable ERRORFILE](#)

[Option -WmsgFb](#)

[Option -WmsgFi](#)

[Option -WmsgFonp](#)

[Option -WmsgFob](#)

-WmsgFonf: Message Format for no File Information

Group

MESSAGE

Scope

Function

Syntax

"-WmsgFonf"<string>

Arguments

<string>: format string (See below)

Default

-WmsgFonf"%K %d: %m\n"

Defines

None

Pragmas

None

Description

Sometimes there is no file information available for a message (e.g. if a message not related to a specific file). Then the message format string defined by MESSAGE is used. [Table 1.26](#) lists the supported formats.

Table 1.26 Message Format Specifiers

Format	Description	Example
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815

Table 1.26 Message Format Specifiers

Format	Description	Example
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space	
%'	A ' if the filename, the path or the extension contains a space	

Example

`-WmsgFonf "%k %d: %m\n"`

Produces a message in following format:

`information L10324: Linking successful`

See also

[Environment variable ERRORFILE](#)

[Option -WmsgFb](#)

[Option -WmsgFi](#)

[Option -WmsgFonp](#)

[Option -WmsgFoi](#)

-WmsgFonp: Message Format for no Position Information

Group

MESSAGE

Scope

Function

Syntax

"-WmsgFonp"<string>

Arguments

<string>: format string (See below)

Default

-WmsgFonp"%%"%f%e%": %K %d: %m\n"

Defines

None

Pragmas

None

Description

Sometimes there is no position information available for a message (e.g. if a message not related to a certain position). Then the message format string defined by MESSAGE is used. [Table 1.27](#) lists the supported formats.

Table 1.27 Message Format Specifiers

Format	Description	Example
%K	Uppercase kind	ERROR
%k	Lowercase kind	error
%d	Number	C1815

Table 1.27 Message Format Specifiers

Format	Description	Example
%m	Message	text
%%	Percent	%
\n	New line	
%"	A " if the filename, if the path or the extension contains a space	
%'	A ' if the filename, the path, or the extension contains a space	

Example

`-WmsgFonf "%k %d: %m\n"`

Produces a message in following format:

information L10324: Linking successful

See also

[Environment variable ERRORFILE](#)

[Option -WmsgFb](#)

[Option -WmsgFi](#)

[Option -WmsgFonp](#)

[Option -WmsgFoi](#)

-WmsgNe: Number of Error Messages

Group

MESSAGE

Scope

Compilation Unit

Syntax

"-WmsgNe" <number>

Arguments

<number>: Maximum number of error messages

Default

50

Defines

None

Pragmas

None

Description

This option sets the number of error messages that are to be displayed while the Compiler is processing.

NOTE	Subsequent error messages which depend upon a previous error message may not process correctly.
-------------	-------------------------------------------------------------------------------------------------

Example

`-WmsgNe2`

Stops compilation after two error messages

See also

[Option -WmsgNi](#)
[Option -WmsgNw](#)

-WmsgNi: Number of Information Messages

Group

MESSAGE

Scope

Compilation Unit

Syntax

"-WmsgNi" <number>

Arguments

<number>: Maximum number of information messages

Default

50

Defines

None

Pragmas

None

Description

This option sets the amount of information messages that are logged.

Example

-WmsgNi10

Ten information messages logged

See also

[Option -WmsgNe](#)
[Option -WmsgNw](#)

-WmsgNu: Disable User Messages

Group

MESSAGE

Scope

None

Syntax

`"-WmsgNu" ["=" {"a" | "b" | "c" | "d"}]`

Arguments

- “a”: Disable messages about include files
- “b”: Disable messages about reading files
- “c”: Disable messages about generated files
- “d”: Disable messages about processing statistics
- “e”: Disable informal messages

Default

None

Defines

None

Pragmas

None

Description

The application produces messages that are not in the following normal message categories: WARNING, INFORMATION, WRROR, FATAL. This option disables messages that are not in the normal message category by reducing the amount of messages, and simplifying the error parsing of other tools.

- “a”: Disables the application from generating information about all included files.
- “b”: Disables messages about reading files (e.g. the files used as input) are disabled.

- “c”: Disables messages informing about generated files.
- “d”: Disables information about statistics (e.g. code size, RAM/ROM usage and so on).
- “e”: Disables informal messages (e.g. memory model, floating point format, ...) .

NOTE Depending on the application, the Compiler may not recognize all suboptions. In this case they are ignored for compatibility.

Example

`-WmsgNu=c`

See also

None

-WmsgNw: Number of Warning Messages

Group

MESSAGE

Scope

Compilation Unit

Syntax

"-WmsgNw" <number>

Arguments

<number>: Maximum number of warning messages

Default

50

Defines

None

Pragmas

None

Description

This option sets the number of warning messages.

Example

-WmsgNw15

Fifteen warning messages logged

See also

[Option -WmsgNe](#)
[Option -WmsgNi](#)

-WmsgSd: Setting a Message to Disable

Group

MESSAGE

Scope

Function

Syntax

"-WmsgSd" <number>

Arguments

<number>: Message number to be disabled, e.g. 1801

Default

None

Defines

None

Pragmas

None

Description

This option disables message from appearing in the error output.

This option cannot be used in a [pragma OPTION](#). Use this option only with [pragma MESSAGE](#).

Example

-WmsgSd1801

Disables message for implicit parameter declaration

See also

[Option -WmsgSi](#)
[Option -WmsgSw](#)

[Option -WmsgSe](#)
[Pragma MESSAGE](#)

-WmsgSe: Setting a Message to Error

Group

MESSAGE

Scope

Function

Syntax

"-WmsgSe" <number>

Arguments

<number>: Message number to be an error, e.g. 1853

Default

None

Defines

None

Pragmas

None

Description

This option changes a message to an error message.

This option cannot be used in a [pragma OPTION](#). Use this option only with [pragma MESSAGE](#).

Example

```
COMPOTIONS=-WmsgSe1853
```

See also

[Option -WmsgSd](#)
[Option -WmsgSi](#)

[Option -WmsgSw](#)
[Pragma MESSAGE](#)

-WmsgSi: Setting a Message to Information

Group

MESSAGE

Scope

Function

Syntax

"-WmsgSi" <number>

Arguments

<number>: Message number to be an information, e.g. 1853

Default

None

Defines

None

Pragmas

None

Description

This option sets a message to an information message.

This option cannot be used with [pragma OPTION](#). Use this option only with [pragma MESSAGE](#).

Example

-WmsgSi1853

See also

[Option -WmsgSd](#)
[Option -WmsgSw](#)

[Option -WmsgSe](#)
[Pragma MESSAGE](#)

-WmsgSw: Setting a Message to Warning

Group

MESSAGE

Scope

Function

Syntax

"-WmsgSw" <number>

Arguments

<number>: Error number to be a warning, e.g. 2901

Default

None

Defines

None

Pragmas

None

Description

This option sets a message to a warning message.

This option cannot be used with [pragma OPTION](#). Use this option only with [pragma MESSAGE](#).

Example

-WmsgSw2901

See also

[Option -WmsgSd](#)
[Option -WmsgSi](#)

[Option -WmsgSe](#)
[Pragma MESSAGE](#)

-WOutFile: Create Error Listing File

Group

MESSAGE

Scope

Compilation Unit

Syntax

"-WOutFile" ("On" | "Off")

Arguments

None

Default

Error listing file is created

Defines

None

Pragmas

None

Description

This option controls whether an error listing file should be created. The error listing file contains a list of all messages and errors that are created during processing. It is possible to obtain this feedback without an explicit file since the text error feedback can now also be handled with pipes to the calling application. The name of the listing file is controlled by the [ERRORFILE](#) environment variable.

Example

-WOutFileOn

Error file is created as specified with [ERRORFILE](#)

-WOutFileOff

No error file created

See also

[Option -WErrFile](#)
[Option -WStdout](#)

-Wpd: Error for Implicit Parameter Declaration

Group

MESSAGE

Scope

Function

Syntax

"-Wpd"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

This option prompts the Compiler to issues an *ERROR* message instead of a *WARNING* message when an implicit declaration is encountered. This occurs if the Compiler does not have a prototype for the called function.

This option helps to prevent parameter passing errors, which can only be detected at runtime. It requires that each function that is called, is prototyped before use. The correct ANSI behavior is to assume that parameters are correct for the stated call.

This option is the same as using -WmsgSe1801.

Example

```
-Wpd  
main() {  
    char a, b;  
    func(a, b); // <- Error here  
}  
func(a, b, c)  
    char a, b, c;  
{  
    ...  
}
```

See also

[Message C1801](#)
[Option -WmsgSe](#)
[Implicit Parameter Declaration in the Front End](#)

-WStdout: Write to Standard Output

Group

MESSAGE

Scope

Compilation Unit

Syntax

"-WStdout" ("On" | "Off")

Arguments

None

Default

Output is written to stdout

Defines

None

Pragmas

None

Description

The usual standard streams are available with Windows applications. Text written into them does not appear anywhere unless explicitly requested by the calling application. This option determines if error file text to error file is also written into the stdout file.

Example

-WStdoutOn

All messages written to stdout

-WErrFileOff

Nothing written to stdout

See also

[Option -WErrFile](#)
[Option -WOutFile](#)

-W1: No Information Messages

Group

MESSAGE

Scope

Function

Syntax

"-W1"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Inhibits printing INFORMATION messages. Only WARNINGS and ERROR messages are generated.

Example

-W1

See also

[Option -WmsgNi](#)

-W2: No Information and Warning Messages

Group

MESSAGE

Scope

Function

Syntax

"-W2"

Arguments

None

Default

None

Defines

None

Pragmas

None

Description

Suppresses all messages of type INFORMATION and WARNING. Only ERRORS are generated.

Example

-W2

See also

[Option -WmsgNi](#)

[Option -WmsgNw](#)

Using the Compiler

Compiler Options

Compiler Predefined Macros

The ANSI standard for the C language requires the Compiler to predefine a couple of macros. The Compiler provides the predefined macros listed in [Table 1.28](#).

Table 1.28 Macros Defined by the Compiler

Macro	Description
<code>__LINE__</code>	Line number in the current source file
<code>__FILE__</code>	Name of the source file it appears in
<code>__DATE__</code>	The date of compilation as a string
<code>__TIME__</code>	The time of compilation as a string
<code>__STDC__</code>	Set to 1 if the -Ansi command line option has been given. Otherwise, additional keywords are accepted (not in ANSI standard).

The following tables lists all Compiler defines with their associated names and options.

NOTE	If these macros do not have a value, the Compiler treats them as if they had been defined as shown: #define <code>__HIWARE__</code>
-------------	-------------------------------------------------------------------------------------------------------------------------------------

It is also possible to log all Compiler predefined defines to a file using the [-Ldf](#) option.

Compiler Vendor Defines

[Table 1.29](#) shows the defines identifying the Compiler vendor. Compilers in the USA may also be sold by ARCHIMEDES.

Table 1.29 Compiler Vendor Identification Defines

Name	Defined
<code>__HIWARE__</code>	always
<code>__MWERKS__</code>	always, set to 1
<code>__ARCHIMEDES__</code>	always

Product Defines

[Table 1.30](#) shows the Defines identifying the Compiler. The Compiler may either be a HI-CROSS Compiler (V2.7.x), a Smile~Line Compiler (V3.0.x) or a HI-CROSS+ Compiler (V5.0.x).

Table 1.30 Compiler Identification Defines

Name	Defined
<code>__PRODUCT_HICROSS__</code>	defined for V2.7 Compilers
<code>__PRODUCT_SMILE_LINE__</code>	defined for V3.0 Compilers
<code>__PRODUCT_HICROSS_PLUS__</code>	defined for V5.0 Compilers
<code>__DEMO_MODE__</code>	defined if the Compiler is running in demo mode
<code>__VERSION__</code>	defined and contains the version number, e.g. it is set to 5013 for a Compiler V5.0.13, or set to 3140 for a Compiler V3.1.40

Data Allocation Defines

The Compiler provides two macros that define how data is organized in memory: Little Endian (least significant byte first in memory) or Big Endian (most significant byte first in memory). The ‘Intel World’ uses Little Endian and the ‘Non-Intel World’ uses Big Endian.

The Compiler provides the “endian” macros listed in [Table 1.31](#).

Table 1.31 Compiler Macros for Defining “Endianness”

Name	Defined
<code>__LITTLE_ENDIAN__</code>	defined if the Compiler allocates in Little Endian order
<code>__BIG_ENDIAN__</code>	defined if the Compiler allocates in Big Endian order

Following example illustrates the difference:

```
unsigned long L = 0x87654321;
unsigned short s = *(unsigned short*)&L; //BE: 0x8765, LE: 0x4321
unsigned char c = *(unsigned char*)&L; //BE: 0x87, LE: 0x21
```

Various Defines for Compiler Option Settings

The following table lists Defines for miscellaneous compiler option settings.

Table 1.32 Defines for Miscellaneous Compiler Option Settings

Name	Defined
__cplusplus	-C++f, -C++e, -C++c
__STDC__	-Ansi
__TRIGRAPH__	-Ci
__CNI__	-Cni
__OPTIMIZE_FOR_TIME__	-Ot
__OPTIMIZE_FOR_SIZE__	-Os

Option Checking in C Code

You can also check the source to determine if an option is active. The EBNF syntax is:

```
OptionActive = "__OPTION_ACTIVE__" "( " string " )".
```

The above is used in the preprocessor and in C code, as shown:

```
#if __OPTION_ACTIVE__( "-W2" )
    // option -W2 is set
#endif

void main(void) {
    int i;
    if (__OPTION_ACTIVE__( "-or" )) {
        i=2;
    }
}
```

You can check all preprocessor-valid options (e.g. options given at the command line, via default.env or project.ini, but not options added with the [#pragma OPTION](#)). You perform the same check in C code using -Odocf and [#pragma OPTION](#).

As a parameter, only the option itself is tested and not a specific argument of an option.

For example:

```
#if __OPTION_ACTIVE__("-D") /* true if any -d option given */
#ifndef __OPTION_ACTIVE__("-DABS") /* not allowed */
```

To check for a specific define use:

```
#if defined(ABS)
```

If the specified option cannot be checked to determine if it is active (i.e. options that no longer exist), the message “[C1439: illegal pragma __OPTION_ACTIVE__](#)” is issued.

ANSI-C Standard Types 'size_t', 'wchar_t' and 'ptrdiff_t' Defines

ANSI provides some standard defines in 'stddef.h' to deal with the implementation of defined object sizes.

[Listing 1.3](#) show part of the contents of stdtypes.h (included from stddef.h).

Listing 1.3 Type Definitions of ANSI-C Standard Types

```
/* size_t: defines the maximum object size type */
#ifndef __SIZE_T_DEFINED__
#define __SIZE_T_DEFINED__

#if defined(__SIZE_T_IS_UCHAR__)
    typedef unsigned char size_t;
#elif defined(__SIZE_T_IS USHORT__)
    typedef unsigned short size_t;
#elif defined(__SIZE_T_IS_UINT__)
    typedef unsigned int size_t;
#elif defined(__SIZE_T_IS ULONG__)
    typedef unsigned long size_t;
#else
    #error "illegal size_t type"
#endif

/* ptrdiff_t: defines the maximum pointer difference type */
#ifndef __PTRDIFF_T_DEFINED__
#define __PTRDIFF_T_DEFINED__

#if defined(__PTRDIFF_T_IS_CHAR__)
    typedef signed char ptrdiff_t;
```

```

#define _PTRDIFF_T_IS_SHORT_
    typedef signed short ptrdiff_t;
#define _PTRDIFF_T_IS_INT_
    typedef signed int ptrdiff_t;
#define _PTRDIFF_T_IS_LONG_
    typedef signed long ptrdiff_t;
#else
    #error "illegal ptrdiff_t type"
#endif
/* wchar_t: defines the type of wide character */
#if defined(_WCHAR_T_IS_UCHAR_)
    typedef unsigned char wchar_t;
#elif defined(_WCHAR_T_IS USHORT_)
    typedef unsigned short wchar_t;
#elif defined(_WCHAR_T_IS_UINT_)
    typedef unsigned int wchar_t;
#elif defined(_WCHAR_T_IS ULONG_)
    typedef unsigned long wchar_t;
#else
    #error "illegal wchar_t type"
#endif

```

[Table 1.33](#) lists defines that deal with other possible implementations:

Table 1.33 Defines for Other Implementations

Macro	Description
<code>_SIZE_T_IS_UCHAR_</code>	Defined if the Compiler expects size_t in stddef.h to be 'unsigned char'.
<code>_SIZE_T_IS USHORT_</code>	Defined if the Compiler expects size_t in stddef.h to be 'unsigned short'.
<code>_SIZE_T_IS_UINT_</code>	Defined if the Compiler expects size_t in stddef.h to be 'unsigned int'.
<code>_SIZE_T_IS ULONG_</code>	Defined if the Compiler expects size_t in stddef.h to be 'unsigned long'.
<code>_WCHAR_T_IS_UCHAR_</code>	Defined if the Compiler expects wchar_t in stddef.h to be 'unsigned char'.
<code>_WCHAR_T_IS USHORT_</code>	Defined if the Compiler expects wchar_t in stddef.h to be 'unsigned short'.
<code>_WCHAR_T_IS_UINT_</code>	Defined if the Compiler expects wchar_t in stddef.h to be 'unsigned int'.

Table 1.33 Defines for Other Implementations

Macro	Description
<code>__WCHAR_T_IS ULONG__</code>	Defined if the Compiler expects wchar_t in stddef.h to be 'unsigned long'.
<code>__PTRDIFF_T_IS CHAR__</code>	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'char'.
<code>__PTRDIFF_T_IS SHORT__</code>	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'short'.
<code>__PTRDIFF_T_IS INT__</code>	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'int'.
<code>__PTRDIFF_T_IS LONG__</code>	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'long'

The following tables show the default settings of the ANSI-C Compiler standard types size_t and ptrdiff_t.

Macros for XGATE

[Table 1.34](#) shows the settings for the XGATE target:

Table 1.34 XGATE Compiler Defines

size_t Macro	Defined
<code>__SIZE_T_IS UCHAR__</code>	never
<code>__SIZE_T_IS USHORT__</code>	never
<code>__SIZE_T_IS UINT__</code>	always
<code>__SIZE_T_IS ULONG__</code>	never

Table 1.35 XGATE Compiler Pointer Difference Macros

ptrdiff_t Macro	Defined
<code>__PTRDIFF_T_IS CHAR__</code>	never
<code>__PTRDIFF_T_IS SHORT__</code>	never

Table 1.35 XGATE Compiler Pointer Difference Macros

ptrdiff_t Macro	Defined
<code>__PTRDIFF_T_IS_INT__</code>	always
<code>__PTRDIFF_T_IS_LONG__</code>	never

Division and Modulus

To ensure that the results of the "/" and "%" operators are defined correctly for signed arithmetic operations, both operands must be defined positive (Refer to the backend chapter). It is implementation-defined if the result is negative or positive when one of the operands is defined negative. This is illustrated in the following example:

```
#ifdef __MODULO_IS_POSITIVE__
    22 / 7 == 3;    22 % 7 == 1
    22 / -7 == -3; 22 % -7 == 1
    -22 / 7 == -4; -22 % 7 == 6
    -22 / -7 == 4; -22 % -7 == 6
#else
    22 / 7 == 3;    22 % 7 == +1
    22 / -7 == -3; 22 % -7 == +1
    -22 / 7 == -3; -22 % 7 == -1
    -22 / -7 == 3; -22 % -7 == -1
#endif
```

The following sections show how it is implemented in a back end.

Object File Format Defines

The Compiler defines some macros to identify the format (mainly used in the startup code if it is object file specific), depending on the specified object file format option. [Table 1.36](#) lists these defines.

Table 1.36 Object File Format Defines

Name	Defined
<code>__HIWARE_OBJECT_FILE_FORMAT__</code>	-Fh
<code>__ELF_OBJECT_FILE_FORMAT__</code>	-F1 , -F2

Bit Field Defines

Bit Field Allocation

The Compiler provides six predefined macros to distinguish between the different allocations:

```
__BITFIELD_MSBIT_FIRST__ /* defined if bitfield  
allocation starts with MSBit */  
  
__BITFIELD_LSBIT_FIRST__ /* defined if bitfield  
allocation starts with LSBit */  
  
__BITFIELD_MSBYTE_FIRST__ /* allocation of bytes starts  
with MSByte */  
  
__BITFIELD_LSBYTE_FIRST__ /* allocation of bytes starts  
with LSByte */  
  
__BITFIELD_MSWORD_FIRST__ /* defined if bitfield  
allocation starts with MSWord */  
  
__BITFIELD_LSWORD_FIRST__ /* defined if bitfield  
allocation starts with LSWord */
```

Using the above-listed defines, you can write compatible code over different Compiler vendors even if the bit field allocation differs. Note that the allocation order of bit fields is important. For example:

```
struct {  
    /* Memory layout of I/O port:  
        MSB                                     LSB  
        BITA | CCR | DIR | DATA | DDR2  
        1     1     1     4     1  
    */  
  
#ifdef __BITFIELD_MSBIT_FIRST__  
    unsigned int BITA:1;  
    unsigned int CCR :1;  
    unsigned int DIR :1;  
    unsigned int DATA:4;  
    unsigned int DDR2:1;  
#elif defined(__BITFIELD_LSBIT_FIRST__)  
    unsigned int DDR2:1;  
    unsigned int DATA:4;  
    unsigned int DIR :1;  
    unsigned int CCR :1;  
    unsigned int BITA:1;
```

```
#else
    #error "undefined bit field allocation strategy!"
#endif
} MyIOport;
```

If the basic allocation unit for bitfields in the Compiler is a byte, the allocation of memory for bitfields is always from the most significant BYTE to the least significant BYTE. For example, `__BITFIELD_MSBYTE_FIRST__` is defined as shown below:

```
/* example for __BITFIELD_MSBYTE_FIRST__ */
struct {
    unsigned char a:8;
    unsigned char b:3;
    unsigned char c:5;
} MyIOport2;

/* LSBIT_FIRST      */ /* MSBIT_FIRST      */
/* MSByte  LSByte   */ /* MSByte  LSByte   */
/* aaaaaaaaaa ccccccbbb */ /* aaaaaaaaaa bbbcccccc */
```

NOTE

There is no standard way to allocate bit fields. Allocation may vary from compiler to compiler even for the same target. Using bit fields for I/O register access to is non-portable and, for the masking involved in unpacking individual fields, inefficient. It is recommended to use regular bit-and (`&`) and bit-or (`|`) operations for I/O port access.

Bit Field Type Reduction

The Compiler provides two predefined macros for enabled/disabled type size reduction. With type size reduction enabled, the Compiler is free to reduce the type of a bit field. For example, if the size of a bit field is 3, the Compiler uses the `char` type.

```
__BITFIELD_TYPE_SIZE_REDUCTION__ /* defined if Type Size
Reduction is enabled */

__BITFIELD_NO_TYPE_SIZE_REDUCTION__ /* defined if Type Size
Reduction is disabled */
```

It is possible to write compatible code over different Compiler vendors and to get optimized bit fields:

```
struct{
    long b1:4;
```

```
    long b2:4;
} myBitfield;

31          7 3 0
-----
| #####|#####|#####|b2|b1| -BfaTSRoff
-----

7      3      0
-----
| b2 | b1 | -BfaTSRon
-----
```

Sign of Plain Bitfields

For some architectures, the sign of a plain bitfield does not follow standard rules.
Normally for:

```
struct _bits {
    int myBits:3;
} bits;
```

the ‘myBits’ is signed, because plain ‘int’ is also signed. To implement it as an unsigned bitfield, use the following code:

```
struct _bits {
    unsigned int myBits:3;
} bits;
```

However, some architectures need to overwrite this behavior to be compliant to their EABI (Embedded Application Binary Interface). Under those circumstances, the [option -T](#) (if supported) is used. The option affects the following defines:

```
__PLAIN_BITFIELD_IS_SIGNED__ /* defined if plain bitfield
                                is signed */
__PLAIN_BITFIELD_IS_UNSIGNED__ /* defined if plain bitfield
                                is unsigned */
```

Macros for XGATE

[Table 1.37](#) identifies the implementation in the Back End.

Table 1.37 XGATE Compiler—Back End Macros

Name	Defined
<code>__BITFIELD_MSBIT_FIRST__</code>	-BfaBMS
<code>__BITFIELD_LSBIT_FIRST__</code>	-BfaBLS
<code>__BITFIELD_MSBYTE_FIRST__</code>	always
<code>__BITFIELD_LSBYTE_FIRST__</code>	never
<code>__BITFIELD_MSWORD_FIRST__</code>	always
<code>__BITFIELD_LSWORD_FIRST__</code>	never
<code>__BITFIELD_TYPE_SIZE_REDUCTION__</code>	-BfaTSRon
<code>__BITFIELD_NO_TYPE_SIZE_REDUCTION__</code>	-BfaTSRoff
<code>__PLAIN_BITFIELD_IS_SIGNED__</code>	always
<code>__PLAIN_BITFIELD_IS_UNSIGNED__</code>	never

Type Information Defines

The Flexible Type Management sets the defines to identify the type sizes. [Table 1.38](#) lists these defines.

Table 1.38 Type Information Defines

Name	Defined
<code>__CHAR_IS_SIGNED__</code>	see option -T or Back End
<code>__CHAR_IS_UNSIGNED__</code>	see option -T or Back End
<code>__CHAR_IS_8BIT__</code>	see option -T or Back End
<code>__CHAR_IS_16BIT__</code>	see option -T or Back End
<code>__CHAR_IS_32BIT__</code>	see option -T or Back End

Table 1.38 Type Information Defines

Name	Defined
<code>__CHAR_IS_64BIT__</code>	see option -T or Back End
<code>__SHORT_IS_8BIT__</code>	see option -T or Back End
<code>__SHORT_IS_16BIT__</code>	see option -T or Back End
<code>__SHORT_IS_32BIT__</code>	see option -T or Back End
<code>__SHORT_IS_64BIT__</code>	see option -T or Back End
<code>__INT_IS_8BIT__</code>	see option -T or Back End
<code>__INT_IS_16BIT__</code>	see option -T or Back End
<code>__INT_IS_32BIT__</code>	see option -T or Back End
<code>__INT_IS_64BIT__</code>	see option -T or Back End
<code>__ENUM_IS_8BIT__</code>	see option -T or Back End
<code>__ENUM_IS_SIGNED__</code>	see option -T or Back End
<code>__ENUM_IS_UNSIGNED__</code>	see option -T or Back End
<code>__ENUM_IS_16BIT__</code>	see option -T or Back End
<code>__ENUM_IS_32BIT__</code>	see option -T or Back End
<code>__ENUM_IS_64BIT__</code>	see option -T or Back End
<code>__LONG_IS_8BIT__</code>	see option -T or Back End
<code>__LONG_IS_16BIT__</code>	see option -T or Back End
<code>__LONG_IS_32BIT__</code>	see option -T or Back End
<code>__LONG_IS_64BIT__</code>	see option -T or Back End
<code>__LONG_LONG_IS_8BIT__</code>	see option -T or Back End
<code>__LONG_LONG_IS_16BIT__</code>	see option -T or Back End
<code>__LONG_LONG_IS_32BIT__</code>	see option -T or Back End
<code>__LONG_LONG_IS_64BIT__</code>	see option -T or Back End
<code>__FLOAT_IS_IEEE32__</code>	see option -T or Back End
<code>__FLOAT_IS_IEEE64__</code>	see option -T or Back End
<code>__FLOAT_IS_DSP__</code>	see option -T or Back End

Table 1.38 Type Information Defines

Name	Defined
<code>__DOUBLE_IS_IEEE32__</code>	see option -T or Back End
<code>__DOUBLE_IS_IEEE64__</code>	see option -T or Back End
<code>__DOUBLE_IS_DSP__</code>	see option -T or Back End
<code>__LONG_DOUBLE_IS_IEEE32__</code>	see option -T or Back End
<code>__LONG_DOUBLE_IS_IEEE64__</code>	see option -T or Back End
<code>__LONG_DOUBLE_IS_DSP__</code>	see option -T or Back End
<code>__LONG_LONG_DOUBLE_IS_IEEE32__</code>	see option -T or Back End
<code>__LONG_LONG_DOUBLE_IS_IEEE64__</code>	see option -T or Back End
<code>__LONG_LONG_DOUBLE_IS_DSP__</code>	see option -T or Back End
<code>__VTAB_DELTA_IS_8BIT__</code>	see option -T
<code>__VTAB_DELTA_IS_16BIT__</code>	see option -T
<code>__VTAB_DELTA_IS_32BIT__</code>	see option -T
<code>__VTAB_DELTA_IS_64BIT__</code>	see option -T
<code>__PLAIN_BITFIELD_IS_SIGNED__</code>	see option -T or Back End
<code>__PLAIN_BITFIELD_IS_UNSIGNED__</code>	see option -T or Back End

Freescale XGATE Specific Defines

[Table 1.39](#) identifies specific implementations in the Back End.

Table 1.39 XGATE Back End Defines

Name	Defined
<code>__XGATE__</code>	always
<code>__NO_RECURSION__</code>	never
<code>__PTR_SIZE_1__</code>	never
<code>__PTR_SIZE_2__</code>	always
<code>__PTR_SIZE_3__</code>	never

Table 1.39 XGATE Back End Defines

Name	Defined
__PTR_SIZE_4__	never
__OPTIMIZE_REG__	never

Compiler Pragmas

A pragma defines how information is passed from the Compiler Front End to the Compiler Back End, without affecting the parser. In the Compiler, the effect of a pragma on code generation starts at the point of its definition, and ends with the end of the next function. Exceptions to this rule are the pragmas [#pragma ONCE](#) and [#pragma NO_STRING_CONSTR](#), which are valid for one file.

The syntax of a pragma is:

```
#pragma pragma_name [optional_arguments]
```

The optional_arguments value depends on the pragma that you use. Some pragmas do not take arguments.

NOTE	A pragma directive accepts a single pragma with optional arguments. Do not place more than one pragma name in a pragma directive. The following example is not correct syntax: <code>#pragma ONCE NO_STRING_CONSTR</code>
-------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

This is an invalid directive.

The following section describes all of the pragmas that affect the Front End. All other pragmas affect only the code generation process and are described in the Back End section.

Pragma Details

This section describes each Compiler-available pragma. The pragmas are listed in alphabetical order and are divided into separate tables. [Table 1.40](#) lists and defines the topics that appear in the description of each pragma.

Table 1.40 Pragma Documentation Topics

Topic	Description
Scope	Scope of pragma where it is valid. See table below.
Syntax	Specifies the syntax of the pragma in a EBNF format.
Synonym	Lists a synonym for the pragma, none if a synonym does not exist.
Arguments	Describes and lists optional and required arguments for the pragma.
Default	Shows the default setting for the pragma or none.

Table 1.40 Pragma Documentation Topics

Topic	Description
Description	Provides a detailed description of the pragma and how to use it.
Example	Gives an example of usage, and effects of the pragma where possible.
See also	Names related sections.

[Table 1.41](#) is an overview of the different scopes of pragmas.

Table 1.41 Definition of Items That Can Appear in a Pragma's Scope Topic

Scope	Description
File	The pragma is valid from the current position until the end of the source file. Example: If the pragma is in a header file included from a source file, the pragma is not valid in the source file.
Compilation Unit	The pragma is valid from the current position until the end of the whole compilation unit. Example: If the pragma is in a header file included from a source file, it is valid in the source file too.
Data Definition	The pragma affects only the next data definition. Ensure that you always use a data definition behind this pragma in a header file. If not, the pragma is used for the first data segment in the next header file, or in the main file.
Function Definition	The pragma affects only the next function definition. Ensure that you use this pragma in a header file: The pragma is valid for the first function in each source file where such a header file is included if there is no function definition in the header file.
Next pragma with same name	The pragma is used until the same pragma appears again. If no such pragma follows this one, it is valid until the end of the file.

#pragma CODE_SEG: Code Segment Definition

Scope

Next pragma CODE_SEG

Syntax

"#pragma CODE_SEG" (<Modif> <Name> | "DEFAULT")

Synonym

CODE_SECTION

Arguments

<Modif>: Some of the following strings may be used:

- __DIRECT_SEG (compatibility alias: DIRECT)
- __NEAR_SEG (compatibility alias: NEAR)
- __CODE_SEG (compatibility alias: CODE)
- __FAR_SEG (compatibility alias: FAR)

NOTE

The compatibility alias should not be used in new code. It only exists for backwards compatibility.
Some of the compatibility alias names do conflict with defines found in certain header files. Therefore using them can cause hard to detect problems and is discouraged.

The meaning of these segment modifiers are backend-dependent. Refer to the backend chapter for information on supported modifiers and their definitions.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Refer to the linker manual for details.

Default

DEFAULT

Description

This pragma specifies the function segment it is allocated in. The segment modifiers also specify the function's calling convention. The CODE_SEG pragma sets the current code segment. This segment places all new function definitions. Also, all function declarations get the current code segment, when they occur. The segment modifiers of this segment determine the calling convention.

The CODE_SEG pragma affects function declarations as well as definitions. Ensure that all function declarations and their definitions are in the same segment.

The synonym CODE_SECTION has exactly the same meaning as CODE_SEG.

Example

```
/* in a header file */
#pragma CODE_SEG __FAR_SEG MY_CODE1
extern void f(void);
#pragma CODE_SEG MY_CODE2
extern void h(void);
#pragma CODE_SEG DEFAULT

/* in the corresponding C file : */
#pragma CODE_SEG __FAR_SEG MY_CODE1
void f(void){ /* f has FAR calling convention */
    h(); /* calls h with default calling convention */
}
#pragma CODE_SEG MY_CODE2
void h(void){ /* f has default calling convention */
    f(); /* calls f with FAR calling convention */
}
#pragma CODE_SEG DEFAULT
```

NOTE Not all back ends support a FAR calling convention.

NOTE The calling convention can also be specified with a supported keyword. The default calling convention is chosen with the memory model.

Illegal usages of the pragma:

```
#pragma DATA_SEG DATA1
#pragma CODE_SEG DATA1
/* error: segment name has different types! */

#pragma CODE_SEG DATA1
#pragma CODE_SEG __FAR_SEG DATA1
/* error: segment name has modifiers! */

#pragma CODE_SEG DATA1
void g(void);
#pragma CODE_SEG DEFAULT
void g(void) d) {}
/* error g is declared in different segments */

#pragma CODE_SEG __FAR_SEG DEFAULT
/* error modifiers for DEFAULT segment no allowed */
```

See also

Compiler Backend
[Segmentation Chapter in the front end](#)
Linker Manual
[#pragma CONST_SEG](#)
[#pragma DATA_SEG](#)
[#pragma STRING_SEG](#)
[Compiler option -Cc](#)
[#pragma push](#)

#pragma CONST_SEG: Constant Data Segment Definition

Scope

Next pragma CONST_SEG

Syntax

"#pragma CONST_SEG" (<Modif> <Name> | "DEFAULT")

Synonym

CONST_SECTION

Arguments

<Modif>: Some of the following strings may be used:

- __SHORT_SEG (compatibility alias: SHORT)
- __DIRECT_SEG (compatibility alias: DIRECT)
- __NEAR_SEG (compatibility alias: NEAR)
- __CODE_SEG (compatibility alias: CODE)
- __FAR_SEG (compatibility alias: FAR)

NOTE	The compatibility alias should not be used in new code. It only exists for backwards compatibility. Some of the compatibility alias names do conflict with defines found in certain header files. Therefore using them can cause hard to detect problems and is discouraged.
-------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The segment modifiers are backend-dependent. Refer to the backend chapter to find the supported modifiers and their meaning. The __SHORT_SEG modifier specifies a segment which is accessed with - bit addresses.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Please refer to the linker manual for details.

Default

DEFAULT

Description

This pragma allocates constant variables into a segment. The segment is then located in the link parameter file to specific addresses.

The pragma CONST_SEG sets the current const segment. This segment is places all constant variable declarations. The default segment is set with:

```
#pragma CONST_SEG DEFAULT"
```

Constants are allocated in the current data segment that is defined with [#pragma DATA_SEG](#) in the HIWARE object file format when the option -cc is not specified, and until the first #pragma CONST_SEG occurs in the source. With option [Cc](#), constants are always allocated in constant segments in the ELF object file format, and after the first #pragma CONST_SEG.

The CONST_SEG pragma also affects constant variable declarations as well as definitions. Ensure that all constant variable declarations and definitions are in the same const segment.

Some compiler optimizations assume that objects having the same segment are placed together. Backends supporting banked data, for example, may set the page register only once for two accesses to two different variables in the same segment. This is also the case for the DEFAULT segment. When using a paged access to variables, place one segment on one page in the link parameter file.

When the [#pragma INTO_ROM](#) is active, the current const segment is not used.

The synonym CONST_SECTION has exactly the same meaning as CONST_SEG.

Example

[Listing 1.4](#) shows code the uses the CONST_SEG pragma.

Listing 1.4 Correct Use of the CONST_SEG Pragma

```
/* in a header file */
#pragma CONST_SEG __SHORT_SEG SHORT_CONST_MEMORY
extern const int i_short;
#pragma CONST_SEG CUSTOM_CONST_MEMORY
extern const int j_custom;
#pragma CONST_SEG DEFAULT
```

```
/* in some C file, which includes the header file */

void main(void) {
    int k= i; /* may use short access */
    k= j;
}

/* in the C file defining the constants : */
#pragma CONST_SEG __SHORT_SEG SHORT_CONST_MEMORY
extern const int i_short=7
#pragma CONST_SEG CUSTOM_CONST_MEMORY
extern const int j_custom=8;
#pragma CONST_SEG DEFAULT
```

[Listing 1.5](#) shows code that uses the CONST_SEG pragma *illegally*.

Listing 1.5 Illegal use of the CONST_SEG Pragma

```
#pragma DATA_SEG CONST1
#pragma CONST_SEG CONST1 /* error: segment name has different types ! */

#pragma CONST_SEG C2
#pragma CONST_SEG __SHORT_SEG C2 //error: segment name has modifiers!

#pragma CONST_SEG CONST1
extern int i;
#pragma CONST_SEG DEFAULT
int i; /* error: i is declared in different segments */

#pragma CONST_SEG __SHORT_SEG DEFAULT // error: modifiers for DEFAULT
//                                segment not allowed
```

See also

Compiler Backend
[Segmentation Chapter in the front end](#)
Linker Manual
[#pragma CODE_SEG](#)
[#pragma DATA_SEG](#)
[#pragma STRING_SEG](#)
[Compiler option -Cc](#)
[#pragma INTO_ROM](#)
[#pragma push](#)

#pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing

Scope

Next pragma CREATE_ASM_LISTING

Syntax

"#pragma CREATE_ASM_LISTING" ("ON" | "OFF")

Synonym

None

Arguments

ON: All following defines / objects are emitted

OFF: All following defines / objects are not emitted

Default

OFF

Description

This pragma controls if the following defines or objects are printed into the assembler include file.

A new file is only generated when the [Option -La](#) is specified together with a header file containing a "#pragma CREATE_ASM_LISTING ON".

Example

```
#pragma CREATE_ASM_LISTING ON
extern int i; /* i is accessible from asm code */

#pragma CREATE_ASM_LISTING OFF
extern int j; /* j is only accessible from C code */
```

See also

[Compiler Option -La](#)
[Generating an Assembler Include File](#)

#pragma DATA_SEG: Data Segment Definition

Scope

Next pragma DATA_SEG

Syntax

"#pragma DATA_SEG" (<Modif> <Name> | "DEFAULT").

Synonym

DATA_SECTION.

Arguments

<Modif>: Some of the following strings may be used:

- __SHORT_SEG (compatibility alias: SHORT)
- __DIRECT_SEG (compatibility alias: DIRECT)
- __NEAR_SEG (compatibility alias: NEAR)
- __CODE_SEG (compatibility alias: CODE)
- __FAR_SEG (compatibility alias: FAR)

NOTE	The compatibility alias should not be used in new code. It only exists for backwards compatibility. Some of the compatibility alias names do conflict with defines found in certain header files. Therefore using them can cause hard to detect problems and is discouraged.
-------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The __SHORT_SEG modifier specifies a segment which is accessed with 8 bit addresses. The meaning of these segment modifiers are backend dependent. Read the backend chapter to find the supported modifiers and their meaning.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Please refer to the linker manual for details.

Default

DEFAULT

Description

This pragma allocates variables into a segment. This segment is then located in the link parameter file to specific addresses.

The pragma DATA_SEG sets the current data segment. This segment is used to place all variable declarations. The default segment is set with:

```
#pragma DATA_SEG DEFAULT"
```

Constants are also allocated in the current data segment in the HIWARE object file format when the option -cc is not specified and no "#pragma CONST_SEG" occurred in the source. With option -Cc and in the ELF object file format, constants are not allocated in the data segment.

The pragma DATA_SEG also affects data declarations, as well as definitions. Ensure that all variable declarations and definitions are in the same segment.

Some compiler optimizations assume that objects having the same segment are together. Backends supporting banked data, for example, may set the page register only once if two accesses two different variables in the same segment are done. This is also the case for the DEFAULT segment. When using a paged access to constant variables, put one segment on one page in the link parameter file.

When the #pragma INTO_ROM is active, the current data segment is not used.

The synonym DATA_SECTION has exactly the same meaning as DATA_SEG.

Example

[Listing 1.6](#) shows code that uses the DATA_SEG pragma.

Listing 1.6 Using the DATA_SEG Pragma

```
/* in a header file */
#pragma DATA_SEG __SHORT_SEG SHORT_MEMORY
extern int i_short;
#pragma DATA_SEG CUSTOM_MEMORY
extern int j_custom;
#pragma DATA_SEG DEFAULT

/* in the corresponding C file : */
#pragma DATA_SEG __SHORT_SEG SHORT_MEMORY
```

```
int i_short;
#pragma DATA_SEG CUSTOM_MEMORY
int j_custom;
#pragma DATA_SEG DEFAULT

void main(void) {
    i = 1; /* may use short access */
    j = 5;
}
```

[Listing 1.6](#) shows code that uses the DATA_SEG pragma *illegally*.

Listing 1.7 Illegal use of the DATA_SEG Pragma

```
#pragma DATA_SEG DATA1
#pragma CONST_SEG DATA1 /* error: segment name has different types ! */

#pragma DATA_SEG DATA1
#pragma DATA_SEG __SHORT_SEG DATA1
/* error: segment name has modifiers ! */

#pragma DATA_SEG DATA1
extern int i;
#pragma DATA_SEG DEFAULT
int i; /* error: i is declared in different segments */

#pragma DATA_SEG __SHORT_SEG DEFAULT
/* error: modifiers for DEFAULT segment no allowed */
```

See also

[Compiler Backend](#)
[Segmentation Chapter in the front end](#)
[Linker Manual](#)
[#pragma CODE_SEG](#)
[#pragma CONST_SEG](#)
[#pragma STRING_SEG](#)
[Compiler option -Cc](#)
[#pragma INTO_ROM](#)
[#pragma push](#)

#pragma INLINE: Inline Next Function Definition

Scope

Function Definition

Syntax

"#pragma INLINE"

Synonym

None

Arguments

None

Default

None

Description

This pragma directs the Compiler to inline the next function in the source.

The pragma is the same as using the ‘inline’ keyword in C++ or using the [-O1](#) Compiler option.

Example

```
int i;
#pragma INLINE
static void foo(void) {
    i = 12;
}
void main(void) {
    foo(); // results into 'i = 12;'
}
```

See also

[#pragma NO_INLINE](#)
[Compiler Option -O1](#)
[C++ Front End](#)

#pragma INTO_ROM: Put Next Variable Definition into ROM

Scope

Data Definition

Syntax

"#pragma INTO_ROM"

Synonym

None

Arguments

None

Default

None

Description

This pragma forces the next (non-constant) variable definition to be const (together with the Compiler option [-Cc](#)).

The pragma is active only for the next single variable definition. A following segment pragma (CONST_SEG, DATA_SEG, CODE_SEG) disables the pragma.

NOTE This pragma is only useful for the HIWARE object file format (and not for ELF/DWARF).

NOTE This pragma is to force a non-constant (means normal ‘variable’) object to be recognized as ‘const’ by the compiler. If the variable already is declared as ‘const’ in the source, this pragma is not needed. This pragma was introduced to cheat the constant handling of the compiler, and shall not be used any more. It is supported for legacy reasons only.

Example

[Listing 1.8](#) shows code that uses the INTO_ROM pragma.

Listing 1.8 Using the INTO_ROM Pragma

```
#pragma INTO_ROM
char *const B[ ] = {"hello", "world"};  
  
#pragma INTO_ROM
int constVariable; /* put into ROM_VAR, .rodata */  
  
int other; /* put into default segment */  
  
#pragma INTO_ROM
#pragma DATA_SEG MySeg /* INTO_ROM overwritten! */
int other2; /* put into MySeg */
```

See also

[Compiler option -Cc](#)

See also

None

#pragma LINK_INFO: Pass Information to the Linker

Scope

Function

Syntax

```
#pragma LINK_INFO NAME "CONTENT"
```

Synonym

None

Arguments

NAME: Identifier specific to the purpose of this LINK_INFO.

CONTENT: C style string containing only printable ASCII characters.

Default

None

Description

This pragma instructs the compiler to put the passed name content pair into the ELF file. For the compiler, the used name and its content do have no meaning other than one name can only contain one content. However, multiple pragmas with different NAME's are legal.

For the linker or for the debugger however, the NAME might trigger some special functionality with CONTENT as argument.

The linker collects the CONTENT for every NAME in different object files and issues an message if a different CONTENT is given for different object files.

NOTE

This pragma only works with the ELF object file format.

Example

Apart from extended functionality implemented in the linker or debugger, this feature can also be used for user defined link time consistency checks:

With the following code in a header file used by all compilation units:

```
#ifdef _DEBUG
#pragma LINK_INFO MY_BUILD_ENV DEBUG
#else
#pragma LINK_INFO MY_BUILD_ENV NO_DEBUG
#endif
```

The linker will issue a message if object files built with `_DEBUG` are linked with object files built without it.

See also

None

#pragma LOOP_UNROLL: Force Loop Unrolling

Scope

Function

Syntax

"#pragma LOOP_UNROLL"

Synonym

None

Arguments

None

Default

None

Description

If this pragma is present, loop unrolling is performed for the next function. This is the same as if the option -Cu is set for the following single function.

Example

```
#pragma LOOP_UNROLL
void F(void) {
    for(i=0; i<5; i++) { // unrolling this loop
        ...
    }
}
```

See also

[#pragma NO_LOOP_UNROLL](#)
[Compiler Option -Cu](#)

#pragma mark: Entry in CodeWarrior IDE Function List

Scope

Line

Syntax

"#pragma" "mark" {any text}

Synonym

None

Arguments

None

Default

None

Description

This pragma adds an entry into the function list of the CodeWarrior IDE. It also helps to introduce faster code lookups by providing a menu entry which directly jumps to a code position. With the special “#pragma mark ”, a separator line is inserted.

NOTE

The compiler does not actually handle this pragma. The compiler ignores this pragma. The CodeWarrior IDE scans opened source files for this pragma. It is not necessary to recompile a file when this pragma is changed. The IDE updates its menus instantly.

Example

In the example in [Listing 1.9](#), the pragma accesses declarations and definitions.

Listing 1.9 Using the MARK Pragma

```
#pragma mark local function declarations
static void inc_counter(void);
```

```
static void inc_ref(void);  
  
#pragma mark local variable definitions  
static int counter;  
static int ref;  
  
#pragma mark -  
static void inc_counter(void) {  
    counter++;  
}  
static void inc_ref(void) {  
    ref++;  
}
```

See also

None

#pragma MESSAGE: Message Setting

Scope

Compilation Unit or until next pragma MESSAGE

Syntax

```
"#pragma" "MESSAGE" {"WARNING" | "ERROR" | "INFORMATION" |  
"DISABLE" | "DEFAULT") {<CNUM>}
```

Synonym

None

Arguments

<CNUM>: Number of message to be set in the format C1234

Default

None

Description

Messages are selectively set to an information message, a warning message, a disable message, or an error message.

NOTE

This pragma has no effect for messages which are produced during preprocessing. The reason is that the pragma parsing has to be done during normal source parsing, and not during preprocessing.

NOTE

This pragma (as other pragmas) has to be specified outside of the function scope. E.g. it is not possible to change a message inside a function or for a part of a function.

Example

In the example shown in [Listing 1.10](#), braces () were left out.

Listing 1.10 Using the MESSAGE Pragma

```
/* treat C1412: Not a function call, */
/* address of a function, as error */
#pragma MESSAGE ERROR C1412
void f(void);
void main(void) {
    f; /* () is missing, but still legal in C */
/* ERROR because of pragma MESSAGE */
}
```

See also

- [Option -WmsgSi](#)
- [Option -WmsgSw](#)
- [Option -WmsgSd](#)
- [Option -WmsgSe](#)

#pragma NO_ENTRY: No Entry Code

Scope

Function

Syntax

"#pragma NO_ENTRY"

Synonym

None

Arguments

None

Default

None

Description

This pragma suppresses the generation of the entry code and is useful for inline assembler functions.

The code generated in a function with #pragma NO_ENTRY may not be safe. It is assumed that the user ensures stack use.

NOTE	Not all backends support this pragma. Some still generate entry code even if this pragma is specified.
-------------	--------------------------------------------------------------------------------------------------------

Example

[Listing 1.11](#) shows how to use the NO_ENTRY pragma (along with others) to avoid any generated code by the compiler. All code is written in inline assembler.

Listing 1.11 Blocking Compiler-generated Function Management Instructions

```
#pragma NO_ENTRY  
#pragma NO_EXIT
```

```
#pragma NO_FRAME
#pragma NO_RETURN
void Func0(void) {
    asm /* no code should be written by the compiler.*/
    ...
}
```

See also

[#pragma NO_EXIT](#)
[#pragma NO_FRAME](#)
[#pragma NO_RETURN](#)

#pragma NO_EXIT: No Exit Code

Scope

Function

Syntax

"#pragma NO_EXIT"

Synonym

None

Arguments

None

Default

None

Description

This pragma suppresses generation of the exit code and is useful for inline assembler functions.

The code generated in a function with #pragma NO_ENTRY may not be safe. It is assumed that the user ensures stack usage.

NOTE

Not all backends support this pragma. Some still generate exit code even if this pragma is specified.

Example

[Listing 1.12](#) shows how to use the NO_EXIT pragma (along with others) to avoid any generated code by the compiler. All code is written in inline assembler.

Listing 1.12 Blocking Compiler-generated Function Management Instructions

```
#pragma NO_ENTRY  
#pragma NO_EXIT
```

```
#pragma NO_FRAME
#pragma NO_RETURN
void Func0(void) {
    asm /* no code should be written by the compiler.*/
    ...
}
```

See also

[#pragma NO_ENTRY](#)
[#pragma NO_FRAME](#)
[#pragma NO_RETURN](#)

#pragma NO_FRAME: No Frame Code

Scope

Function

Syntax

"#pragma NO_FRAME"

Synonym

None

Arguments

None

Default

None

Description

This pragma suppresses the generation of frame code and is useful for inline assembler functions.

The code generated in a function with #pragma NO_ENTRY may not be safe. It is assumed that the user ensures stack usage.

NOTE	Not all backends support this pragma. Some still generate frame code even if this pragma is specified.
-------------	--------------------------------------------------------------------------------------------------------

Example

[Listing 1.13](#) shows how to use the NO_FRAME pragma (along with others) to avoid any generated code by the compiler. All code is written in inline assembler.

Listing 1.13 Blocking Compiler-generated Function Management Instructions

```
#pragma NO_ENTRY  
#pragma NO_EXIT
```

```
#pragma NO_FRAME
#pragma NO_RETURN
void Func0(void) {
    asm /* no code should be written by the compiler.*/
    ...
}
```

See also

[#pragma NO_ENTRY](#)
[#pragma NO_EXIT](#)
[#pragma NO_RETURN](#)

#pragma NO_INLINE: Do not Inline next Function Definition

Scope

Function

Syntax

"#pragma NO_INLINE"

Synonym

None

Arguments

None

Default

None

Description

This pragma prevents the Compiler to inline the next function in the source. The pragma is used to avoid to inline a function which would be otherwise inlined because of the -Oi Compiler option.

Example

(With option -Oi)

```
int i;
#pragma NO_INLINE
static void foo(void) {
    i = 12;
}
void main(void) {
    foo(); // call is not inlined
}
```

See also

[#pragma INLINE](#)
[Compiler Option -Oi](#)
[C++ Front End](#)

#pragma NO_LOOP_UNROLL: Disable Loop Unrolling

Scope

Function

Syntax

"#pragma NO_LOOP_UNROLL"

Synonym

None

Arguments

None

Default

None

Description

If this pragma is present, no loop unrolling is performed for the next function definition, even if the command line option -Cu is given.

Example

```
#pragma NO_LOOP_UNROLL
void F(void) {
    for(i=0; i<5; i++) { // loop is NOT unrolled
    ...
}
```

See also

[#pragma LOOP_UNROLL](#)
[Compiler Option -Cu](#)

#pragma NO_RETURN: No Return Instruction

Scope

Function

Syntax

"#pragma NO_RETURN"

Synonym

Arguments

Default

Description

This pragma suppresses the generation of the return instruction (return from subroutine, return from interrupt). This may be useful if you care about the return instruction itself, or if the code has to fall through to the first instruction of the next function.

This pragma does not suppress the generation of the exit code at all (e.g. deallocation of local variables, compiler generated local variables). The pragma suppresses the generation of the return instruction.

NOTE	If this feature is used to fall through to the next function, smart linking has to be switched off in the Linker, because the next function may be not referenced from somewhere else. Additionally be careful that both functions are in a linear segment. To be on the safe side allocate both function into a segment, which does only have a linear memory area.
-------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Example

The example in [Listing 1.14](#) places some functions into a special named segment. All functions in this special code segment have to be called from an operating system every 2 seconds after each other. With the pragma, some functions do not return. They fall directly to the next function to be called, saving code size and execution time.

Listing 1.14 Blocking Compiler-generated Function Return Instructions

```
#pragma CODE_SEG CallEvery2Secs
#pragma NO_RETURN
void Func0(void) {
    /* first function, called from OS */
    ...
} /* fall through!!!! */
#pragma NO_RETURN
void Func1(void) {
    ...
} /* fall through */
...
/* last function has to return, no pragma is used! */
void FuncLast(void) {
    ...
}
```

See also

[#pragma NO_ENTRY](#)
[#pragma NO_EXIT](#)
[#pragma NO_FRAME](#)

#pragma NO_STRING_CONSTR: No String Concatenation During Pre-processing

Scope

Compilation Unit

Syntax

"#pragma NO_STRING_CONSTR"

Synonym

Arguments

Default

Description

This pragma is valid for the rest of the file it appears in. It switches off the special handling of '#' as a string constructor. This is useful if a macro contains inline assembler statements using this character, e.g. for IMMEDIATE values.

Example

The following pseudo assembly code macro shows the use of the pragma. Without the pragma, '#' is handled as string constructor, which is not the desired behavior.

```
#pragma NO_STRING_CONSTR
#define HALT(x)    asm { \
                  LOAD Reg,#3 \
                  HALT x, #255\
                }
```

See also

[Using Immediate Addressing Mode in HLI](#)

#pragma ONCE: Include Once

Scope

File

Syntax

"#pragma ONCE"

Synonym

None

Arguments

None

Default

None

Description

If this pragma appears in a header file, the file is opened and read only once. This increases compilation speed.

Example

#pragma ONCE

See also

[Option -Pio](#)

#pragma OPTION: Additional Options

Scope

Compilation Unit or until next pragma OPTION

Syntax

```
"#pragma" "OPTION" ("ADD" [<Handle>] {<Option>} |"DEL" ({"Handle"} | "ALL")).
```

Synonym

None

Arguments

<Handle>: An identifier - added options can selectively be deleted.

<Option>: A valid option string enclosed in " characters

Default

None

Description

Options are added inside of the source code while compiling a file.

The options given on the command line, or in a configuration file, cannot be changed in any way.

Additional options are added to the current ones with the “ADD” command. A handle may be given optionally.

The “DEL” command either removes all options with a specific handle. It also uses the “ALL” keyword to remove all added options regardless if they have a handle or not. Note that you only can remove options which were added previously with the pragma OPTION ADD.

All keywords, and the handle, are case sensitive.

Restrictions:

- The option [-d](#) (preprocessor definition) is not allowed. Use a “#define” preprocessor directive instead.

- The option [-odocf](#) is not allowed. Specify this option on the command line or in a configuration file instead.
- The Message Setting options [-WmsgSi](#), [-WmsgSw](#), [-WmsgSd](#) and [-WmsgSe](#) have no effect. Use the [pragma MESSAGE](#) instead.
- Only options concerning tasks during code generation are used. Options controlling the preprocessor, for example, have no effect.
- No macros are defined for specific options.
- Only options having function scope may be used.
- The given options must not specify a conflict to any other given option.
- The pragma is not allowed inside of declarations or definitions.

Example

The example in [Listing 1.15](#) shows how to compile only a single function with the additional option [-or](#).

Listing 1.15 Using the OPTION Pragma

```
#pragma OPTION ADD function_main_handle "-or"

int sum(int max) { /* compiled with -or */
    int i, sum=0;
    for (i =0; i < max; i++) {
        sum += i;
    }
    return sum;
}

#pragma OPTION DEL function_main_handle
/* now the same options as before the #pragma */
/* OPTION ADD are active again */
```

The examples in [Listing 1.16](#) shows *illegal* uses of the OPTION pragma.

Listing 1.16 Illegal Uses of the OPTION Pragma

```
#pragma OPTION ADD -or /* ERROR, use "-or" */
#pragma OPTION "-or" /* ERROR, use keyword ADD */
#pragma OPTION ADD "-odocf=\\"-or\\\""
/* ERROR, "-odocf" not allowed in this pragma */

void f(void) {
```

Using the Compiler

Compiler Pragmas

```
#pragma OPTION ADD "-or"
/* ERROR, pragma not allowed inside of declarations */
}
#pragma OPTION ADD "-cni"
#ifndef __CNI__
/* ERROR, macros are not defined for options */
/* added with the pragma */
#endif
```

See also

None

#pragma pop: restore pragma state

Scope

Compilation Unit

Syntax

#pragma pop

Arguments

None.

Default

None

Description

The pragma pop allows to restore a pragma state previously stored with a [#pragma push](#). A typical usage is at the end of a header files to avoid that the header file changes the pragma state.

The pragma pop does currently restore the state of the following pragmas:

```
#pragma CODE_SEG  
#pragma CONST_SEG  
#pragma DATA_SEG  
#pragma STRING_SEG  
#pragma align
```

Example

```
/* example.h */  
ifndef EXAMPLE_H_  
define EXAMPLE_H_  
pragma push  
pragma CODE_SEG EXAMPLE_CODE_SEG  
void ExampleFunction(void);  
pragma pop  
endif /* EXAMPLE_H_ */
```

See also

[#pragma CODE_SEG](#)
[#pragma CONST_SEG](#)
[#pragma DATA_SEG](#)
[#pragma STRING_SEG](#)
[#pragma push](#)

#pragma push: save pragma state

Scope

Compilation Unit

Syntax

#pragma push

Arguments

None.

Default

None

Description

The pragma push allows to store the current pragma state so that it can be restored later on with a [#pragma pop](#). A typical usage is in header files to maintain the pragma state while using local pragmas for the header file content.

The pragma push does currently store the state of the following pragmas:

```
#pragma CODE_SEG  
#pragma CONST_SEG  
#pragma DATA_SEG  
#pragma STRING_SEG  
#pragma align
```

Example

```
/* example.h */  
ifndef EXAMPLE_H_  
define EXAMPLE_H_  
pragma push  
pragma CODE_SEG EXAMPLE_CODE_SEG  
void ExampleFunction(void);  
pragma pop  
endif /* EXAMPLE_H_ */
```

See also

[#pragma CODE_SEG](#)
[#pragma CONST_SEG](#)
[#pragma DATA_SEG](#)
[#pragma STRING_SEG](#)
[#pragma pop](#)

#pragma REALLOC_OBJ: Object Reallocation

Scope

Compilation Unit

Syntax

```
#pragma REALLOC_OBJ "segment" ["objfile"] object qualifier
```

Arguments

qualifier = {("__near", "__far", "__paged", "__namemangle")}

segment: Name of an already existing segment. This name must have been previously used by a segment pragma (DATA_SEG, CODE_SEG, CONST_SEG, STRING_SEG).

objfile: Name of a object file. If specified, the object is assumed to have static linkage and to be defined in objfile. The name must be specified without alteration by the qualifier "__namemangle".

object: Name of the object to be reallocated. Here the name as known to the linker has to be specified. For C++ linkage, names must be specified according to the C++ name mangling.

qualifier: One of the qualifiers mentioned above. Some of the qualifiers are only allowed to backends not supporting a specified qualifier generating this message. With the special "__namemangle" qualifier, the link name is changed so that the name of the reallocated object does not match the usual name. This feature detects when a pragma REALLOC_OBJ is not applied to all uses of one object.

Default

None

Description

This pragma reallocates an object (e.g. affecting its calling convention). This is used by the linker if the linker has to distribute objects over banks/segments in an automatic way (code distribution). The linker is able to generate an include file containing #pragma RALLOC_OBJ to tell the compiler how to change calling conventions for each object. See the Linker manual for details.

Example

```
#pragma REALLOC_OBJ "DISTRIBUTE1" ("evaluate.o")
Eval_Plus __near __namemangle
```

See also

[Message C4204](#)

Linker Manual

#pragma STRING_SEG: String Segment Definition

Scope

Next pragma STRING_SEG

Syntax

"#pragma STRING_SEG" (<Modif> <Name> | "DEFAULT")

Synonym

STRING_SECTION

Arguments

<Modif>: Some of the following strings may be used:

- __DIRECT_SEG (compatibility alias: DIRECT)
- __NEAR_SEG (compatibility alias: NEAR)
- __CODE_SEG (compatibility alias: CODE)
- __FAR_SEG (compatibility alias: FAR)

NOTE

The compatibility alias should not be used in new code. It only exists for backwards compatibility.
Some of the compatibility alias names do conflict with defines found in certain header files. Therefore using them can cause hard to detect problems and is discouraged.

The __SHORT_SEG modifier specifies a segment that accesses using 8 bit addresses. The definitions of these segment modifiers are backend dependent. Read the backend chapter to find the supported modifiers and their definitions.

<Name>: The name of the segment. This name must be used in the link parameter file on the left side of the assignment in the PLACEMENT part. Please refer to the linker manual for details.

Default

DEFAULT.

Description

This pragma allocates strings into a segment. Strings are allocated in the linker segment STRINGS. This pragma allocates strings in special segments. String segments also may have modifiers. This instructs the Compiler to access them in a special way when necessary.

Segments defined with the pragma STRING_SEG are treated by the linker like constant segments defined with a [#pragma CONST_SEG](#), so they are allocated in ROM areas.

The pragma STRING_SEG sets the current string segment. This segment is used to place all newly occurring strings.

NOTE	The linker may support a overlapping allocation of strings. E.g. the allocation of “CDE” inside of the string “ABCDE”, so that both strings together need only six bytes. When putting strings into user-defined segments, the linker may no longer do this optimization. Only use a user-defined string segments when necessary.
-------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

The synonym STRING_SECTION has exactly the same meaning as STRING_SEG.

Example

```
#pragma STRING_SEG STRING_MEMORY
char* p="String1";
void f(char*);
void main(void) {
    f("String2");
}
#pragma STRING_SEG DEFAULT
```

See also

Compiler Backend
[Segmentation Chapter in the front end](#)
Linker Manual
[#pragma CODE_SEG](#)
[#pragma CONST_SEG](#)
[#pragma DATA_SEG](#)
[#pragma push](#)

#pragma TEST_CODE: Check Generated Code

Scope

Function Definition

Syntax

```
"#pragma TEST_CODE" CompOp <Size> {<HashCode>}.
```

CompOp= “==” | “!=” | “<” | “>” | “<=” | “>=”.

Arguments

<Size>: Size of the function to be used with compare operation

<HashCode>: optional value specifying one specific code pattern.

Default

None

Description

This pragma checks the generated code. If the check fails issues the message [C3601](#).

The following parts are tested:

Size of the function:

The compare operator and the size given as arguments are compared with the size of the function.

This feature checks that the compiler generates less code than a given boundary. Or, to be sure that certain code it can also be checked that the compiler produces more code than specified. To only check the hashcode use a condition which is always TRUE, such as “!= 0”.

Hashcode:

The compiler produces a 16-bit hashcode from the produced code of the next function. This hashcode considers:

- The code bytes of the generated functions
- The type, offset and addend of any fixup

To get the hashcode of a certain function, compile the function with an active `#pragma TEST_CODE` which will fail. Then copy the computed hashcode out of the body of the message [C3601](#).

NOTE The code generating by the compiler may change. When the test fails, it is often not sure that the topic chosen to be checked was wrong.

Example:

```
/* check that an empty function is smaller */
/* than 10 bytes */
#pragma TEST_CODE < 10
void main(void) {
}
```

You can also use this pragma to detect when different code is generated:

```
/* if the following pragma fails, check the code. */
/* If the code is OK, add the hashcode to the */
/* list of allowed codes : */
#pragma TEST_CODE != 0 25645 37594
/* check code patterns : */
/* 25645 : shift for *2 */
/* 37594 : mult for *2 */
void main(void) {
    f(2*i);
}
```

See also

[Message C3601](#)

#pragma TRAP_PROC: Mark Function as Interrupt Function

Scope

Function Definition

Syntax

"#pragma TRAP_PROC"

Arguments

See Back End

Default

None

Description

This pragma marks a function to be an interrupt function. Because interrupt functions may need some special entry and/or exit code, this pragma has to be used for interrupt functions.

Do not use this pragma for declarations (e.g. in header files), as the pragma is valid for the next definition.

See Back End for details.

NOTE

Using C++, the Compiler will also ‘name mangling’ the name of an interrupt function. The mangled name has to be specified in the linker parameter file (e.g. MyIntFkt_Fv) if the vector number has not been specified with the function definition. To prevent name mangling, declare the function with ‘extern “C”’

Examples:

extern “C” interrupt void MyResetFkt(void) {...

or

#pragma TRAP_PROC

extern “C” void MyResetFkt(void) {...

Example:

```
#pragma TRAP_PROC
void MyInterrupt(void) {
    ...
}
```

See also

[interrupt keyword](#)

ANSI-C Front End

The Compiler Front End reads the source file(s), does all the syntactic and semantic checking, and produces intermediate representation of the program which then is passed on to the Back End to generate code.

This section discusses features, restrictions and further properties of the ANSI-C Compiler Front End.

Implementation Features

The Compiler provides a series of pragmas instead of introducing additions to the language to support features such as interrupt procedures. The Compiler implements ANSI-C according to the X3J11 standard. The reference document is “*American National Standard for Programming Languages – C*”, ANSI/ISO 9899–1990.

Keywords

auto	break	case	char
const	continue	default	do
double	else	enum	extern
float	for	goto	if
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while

Preprocessor Directives

The Compiler supports the full set of preprocessor directives as required by the ANSI standard:

```
#if, #ifdef, #ifndef, #else, #elif, #endif
#define, #undef
#include
#pragma
#error, #line
```

The preprocessor operators `defined`, `#` and `##` are also supported. There is a special non-ANSI directive `#warning` which is the same as `#error`, but issues only a warning message.

Language Extensions

There is a language extension in the Compiler for ANSI-C. You can use keywords to qualify pointers in order to distinguish them, or to mark interrupt routines.

The Compiler supports the following non-ANSI compliant keywords (see Back End if they are supported and for their semantic):

Pointer Qualifiers

```
__far (alias far)
__near (alias near)
```

Pointer qualifiers be used to distinguish between different pointer types (e.g. for paging). Some of them are also used to specify the calling convention to be used (e.g. if banking is available).

To allow portable programming between different CPUs (or if the target CPU does not support an additional keyword), you can include the defines listed below in the ‘`hidef.h`’ header file.

```
#define far      /* no far keyword supported */
#define near     /* no near keyword supported */
```

Special Keywords

```
__alignof__
__va_sizeof__
__interrupt (alias interrupt)
__asm (aliases __asm and asm)
```

NOTE Not all Back Ends may support the above-listed keywords. See section “Non-ANSI Keywords” in the Back End for more details.

ANSI-C was not designed with embedded controllers in mind. The listed keywords do not conform to ANSI standards. However, they do enable an easy way to achieve good results from code used for embedded applications.

You can use the `_interrupt` keyword to mark functions as interrupt functions, and to link the function to a specified interrupt vector number (not supported by all backends).

Binary Constants (0b)

It is as well possible to use the binary notation for constants instead of hexadecimal constants or normal constants. Note that binary constants are not allowed if [option - Ansi](#) is switched on. Binary constants start with the 0b prefix, followed by a sequence of 0 or 1.

```
#define myBinaryConst 0b01011

int i;

void main(void) {
    i = myBinaryConst;
}
```

Hexadecimal Constants (\$)

It is possible to use Hexadecimal constants inside HLI (High Level Inline) Assembly. E.g. instead of 0x1234 you can use \$1234. Note that this is valid only for inline assembly.

#warning Directive

The #warning directive is used as it is similar to the #error directive.

```
#ifndef MY_MACRO
    #warning "MY_MACRO set to default"
    #define MY_MACRO 1234
#endif
```

Global Variable Address Modifier (@address)

You can assign global variables to specific addresses with the global variable address modifier. These variables are called ‘absolute variables’. They are useful for accessing memory mapped I/O ports and have the following syntax:

```
Declaration = <TypeSpec> <Declarator>
              [ @ <Address> | @ "<Section>" ] [= <Initializer>];
```

<TypeSpec> is the type specifier, e.g. int, char

<Declarator> is the identifier of your global object, e.g. i, glob

<Address> is the absolute address of the object, e.g. 0xff04, 0x00+8

<Initializer> is the value to which the global variable is initialized.

A segment is created for each global object specified with an absolute address. This address must not be inside any address range in the SECTIONS entries of the link parameter file, otherwise there would be a linker error (overlapping segments). If the specified address has a size greater than that used for addressing the default data page, pointers pointing to this global variable must be "__far". An alternate way to assign global variables to specific addresses is:

```
#pragma DATA_SEG [ __SHORT_SEG ] <segment_name>
```

setting the PLACEMENT section in the linker parameter file. An older method of accomplishing this was:

```
<segment_name> INTO READ_ONLY <Address> ;
```

Examples

```
int glob @0x0500 = 10; // ok, the global variable "glob"
                      // is at 0x0500, initialized with 10
void g() @0x40c0; // error (the object is a function)
void f() {
    int i @0x40cc; // error (the object is a local variable)
}
```

Corresponding link parameter file settings (prm-file):

```
/* the address 0x0500 of "glob" must not be in any address
   range of the SECTIONS entries */
SECTIONS
    MY_RAM    = READ_WRITE 0x0800 TO 0x1BFF;
    MY_ROM    = READ_ONLY   0x2000 TO 0xFEFF;
    MY_STACK  = READ_WRITE 0x1C00 TO 0x1FFF;
    MY_IO_SEG= READ_WRITE 0x0400 TO 0x4ff;
END
PLACEMENT
    IO_SEG      INTO MY_IO_SEG;
    DEFAULT_ROM INTO MY_ROM;
```

```
DEFAULT_RAM    INTO   MY_RAM;
SSTACK         INTO   MY_STACK;
END
```

Variable Allocation using @ “SegmentName”

Sometimes it is useful, to have the variable directly allocated in a named segment instead using the #pragma way. Example how to do this:

```
#pragma DATA_SEG __SHORT_SEG tiny
#pragma DATA_SEG not_tiny
#pragma DATA_SEG __SHORT_SEG tiny_b
#pragma DATA_SEG DEFAULT
int i@"tiny";
int j@"not_tiny";
int k@"tiny_b";
```

So with some pragmas in a common header file and with another definition for the macro, it is possible to allocate variables depending on a macro.

```
Declaration = <TypeSpec> <Declarator>
              [@ "<Section>" ] [= <Initializer>];
```

Variables declared and defined with the @“section” syntax behave exactly like variables declared after their respective pragmas.

<TypeSpec> is the type specifier, e.g. int, char

<Declarator> is the identifier of your global object, e.g. i, glob

<Section> is the section name. It should be defined in the link parameter file as well. E.g. "MyDataSection".

<Initializer> is the value to which the global variable is initialized.

The used section name has to be known at the declaration time by a previous section pragma.

Examples

```
#pragma DATA_SEC __SHORT_SEG   MY_SHORT_DATA_SEC
#pragma DATA_SEC                  MY_DATA_SEC
#pragma CONST_SEC                 MY_CONST_SEC
#pragma DATA_SEC    DEFAULT // not necessary,
                      // but good practice
#pragma CONST_SEC    DEFAULT // not necessary,
                      // but good practice
int short_var @"MY_SHORT_DATA_SEC"; // OK, accesses are short
int ext_var @"MY_DATA_SEC" = 10; // OK, goes into
```

```
// MY_DATA_SECT
int def_var; / OK, goes into DEFAULT_RAM
const int cst_var @"MY_CONST_SEC" = 10; //ok, goes into
                                         // MY_CONST_SECT
```

Corresponding link parameter file settings (prm-file):

```
SECTIONS
    MY_ZRAM    = READ_WRITE 0x00F0 TO 0x00FF;
    MY_RAM     = READ_WRITE 0x0100 TO 0x01FF;
    MY_ROM     = READ_ONLY   0x2000 TO 0xFEFF;
    MY_STACK   = READ_WRITE 0x0200 TO 0x03FF;
END
PLACEMENT
    MY_CONST_SEC,DEFAULT_ROM INTO MY_ROM;
    MY_SHORT_DATA_SEC        INTO MY_ZRAM;
    MY_DATA_SEC,  DEFAULT_RAM INTO MY_RAM;
    SSTACK                  INTO MY_STACK;
END
```

Absolute Functions

Sometimes it is useful to call a absolute function (e.g. a special function in ROM). Below is a simple example how this could be done using normal ANSI-C.

```
#define erase ((void*)(void))(0xfc06))
void main(void) {
    erase(); /* call function at address 0xfc06 */
}
```

Absolute Variables and Linking

Special attention is needed if absolute variables are involved in the linker's link process.

If an absolute object is not referenced by your application, the absolute variable is not linked in HIWARE format by default. Instead, it is always linked using the ELF/DWARF format. To force linking, switch off smart linking in the Linker, or using the ENTRIES command in the linker parameter file.

NOTE: Interrupt vector entries are always linked.

The example in [Listing 1.17](#) shows how the linker handles different absolute variables.

Listing 1.17 Linker Handling of Absolute Variables

```
char i;          /* zero out           */
```

```
char j = 1;      /* zero out, copy-down */
const char k = 2;    /* download */
char I@0x10;     /* no zero out! */
char J@0x11 = 1; /* copy down */
const char K@0x12 = 2; /* HIWARE: copy down / ELF: download! */
static   char L@0x13;    /* no zero out! */
static   char M@0x14 = 3; /* copy down */
static const char N@0x15 = 4; /* HIWARE: copy down, ELF: download */

void interrupt 2 MyISRfct(void) {} /* download, always linked! */
/* vector number two is downloaded with &MyISRfct */

void foo(char *p) {} /* download */

void main(void) { /* download */
    foo(&i); foo(&j); foo(&k);
    foo(&I); foo(&J); foo(&K);
    foo(&L); foo(&M); foo(&N);
}
```

Zero out means that the default startup code initializes the variables during startup. Copy down means that the variable is initialized during the default startup. To Download means that the memory is initialized while downloading the application.

The __far Keyword

The keyword far is a synonym for __far, which is not allowed when [option -Ansi](#) is present.

NOTE	Not all Back Ends may support this keyword. See section “Non-ANSI Keywords” in the Back End.
-------------	----------------------------------------------------------------------------------------------

A __far pointer allows access to the whole memory range supported by the processor, not just to the default data page. You can use it to access memory mapped I/O registers that are not on the data page. You can also use it to allocate constant strings in a ROM not on the data page.

The ‘__far’ keyword defines the calling convention for a function. Some backends support special calling conventions which also set a page register when a function is called. This enables you to use more code than the address space can usually accommodate. The special allocation of such functions is not done automatically.

Using the __far Keyword for Pointers

The keyword `__far` is a type qualifier like `const` and is valid only in the context of pointer types and functions. The `__far` keyword (for pointers) always affects the last '*' to its left in a type definition. The declaration of a `__far` pointer to a `__far` pointer to a character is:

```
char * __far * __far p;
```

The following is a declaration of a normal (short) pointer to a `__far` pointer to a character:

```
char * __far * p;
```

NOTE To declare a `__far` pointer, place the `__far` keyword *after* the asterisk:

```
char * __far p;
```

not

```
char __far *p;
```

The second choice is *illegal*.

`__far` and Arrays

The `__far` keyword does not appear in the context of the '*' type constructor in the declaration of an array parameter, as shown:

```
void my_func (char a[37]);
```

Such a declaration specifies a pointer argument. This is equal to:

```
void my_func (char *a);
```

There are two possible uses when declaring such an argument to a `__far` pointer:

```
void my_func (char a[37] __far);
```

or alternately

```
void my_func (char * __far a);
```

In the context of the '[]' type constructor in a direct parameter declaration, the `__far` keyword always affects the first dimension of the array to its left. In the following declaration, parameter `a` has type “`__far` pointer to array of 5 `__far` pointers to `char`”:

```
void my_func (char * __far a[][5] __far);
```

`__far` and `typedef` Names

If the array type has been defined as a `typedef` name as in:

```
typedef int ARRAY[10];
```

then a `__far` parameter declaration is:

```
void my_func (ARRAY __far a);
```

The parameter is a `__far` pointer to the first element of the array. This is equal to:

```
void my_func (int *__far a);
```

It is also equal to the following direct declaration:

```
void my_func (int a[10] __far);
```

It is *not* the same as specifying a `__far` pointer to the array:

```
void my_func (ARRAY *__far a);
```

because `a` has type “`__far` pointer to `ARRAY`” instead of “`__far` pointer to `int`”.

__far and Global Variables

The ‘`__far`’ keyword can also be used for global variables:

```
int __far i;           // ok for global variables
int __far *i;          // ok for global variables
int __far *__far i;    // ok for global variables
```

This forces the Compiler to perform the same addressing mode for this variable as it has been declared in a `_FAR_SEG` segment. Note that for the above variable declarations/definitions, the variables are in the `DEFAULT_DATA` segment if no other data segment is active. Be careful if you mix ‘`__far`’ declarations/definitions within a non-`_FAR_SEG` data segment. Assuming that `_FAR_SEG` segments have ‘extended’ addressing mode and normal segments have ‘direct’ addressing mode, the following two examples clarify this behavior:

```
// ok, consistent declarations
#pragma DATA_SEG MyDirectSeg // use direct addressing mode
int i;      // direct, segment MyDirectSeg
int j;      // direct, segment MyDirectSeg

#pragma DATA_SEG __FAR_SEG MyFarSeg // use extended
addressing mode
int k;      // extended, segment MyFarSeg
int l;      // extended, segment MyFarSeg
int __far m; // extended, segment MyFarSeg

// caution: not consistent!!!!
#pragma DATA_SEG MyDirectSeg // use direct addressing mode
int i;      // direct, segment MyDirectSeg
int j;      // direct, segment MyDirectSeg
int __far k; // extended, segment MyDirectSet
```

```
int __far l; // extended, segment MyDirectSeg
int __far m; // extended, segment MyDirectSeg
```

NOTE The `__far` keyword global variables only affects the access to the variable (addressing mode), and NOT the allocation.

`__far` and C++ Classes

If a member function gets the modifier `__far`, the “this” pointer is a `__far` pointer. This is useful, if an instance, if the owner class of the function is not allocated on the default data page. See [Listing 1.18](#).

Listing 1.18 `__far` Member Functions

```
class A {
public:
    void f_far(void) __far {
        /* __far version of member function A::f() */
    }
    void f(void) {
        /* normal version of member function A::f() */
    }
};

#pragma DATA_SEG MyDirectSeg // use direct addressing mode
A a_normal; // normal instance
#pragma DATA_SEG __FAR_SEG MyFarSeg // use extended addressing mode
A __far a_far; // __far instance
void main(void) {
    a_normal.f(); // call normal version of A::f() for normal instance
    a_far.f_far(); // call __far version of A::f() for __far instance
}
```

`__far` and C++ References

The `__far` modifier is applied to references. This is useful if it is a reference to an object outside of the default data page. For example:

```
int j; // object j allocated outside the default data page
       // (must be specified in the link parameter file)
void f(void) {
    int &__far i = j;
}
```

Using the `__far` Keyword for Functions

A special calling convention is specified for the `__far` keyword. The `__far` keyword is specified in front of the function identifier:

```
void __far f(void);
```

If the function returns a pointer, the `__far` keyword must be written in front of the first asterisk (“`*`”).

```
int __far *f(void);
```

It must, however, be after the `int` and not before it.

For function pointers, many backends assume that the `__far` function pointer is pointing to functions with the `__far` calling convention, even if the calling convention was not specified. Moreover, most backends do not support different function pointer sizes in one compilation unit. The function pointer size is then dependent only upon the memory model. See the backend chapter for details.

Table 1.42 Interpretation of the `__far` Keyword

Declaration	Allowed	Type Description
<code>int __far f();</code>	ok	<code>__far</code> function returning an <code>int</code>
<code>__far int f();</code>	error	
<code>__far f();</code>	ok	<code>__far</code> function returning an <code>int</code>
<code>int __far *f();</code>	ok	<code>__far</code> function returning a pointer to <code>int</code>
<code>int * __far f();</code>	ok	function returning a <code>__far</code> pointer to <code>int</code>
<code>__far int * f();</code>	error	
<code>int __far * __far f();</code>	ok	<code>__far</code> function returning a <code>__far</code> pointer to <code>int</code>
<code>int __far i;</code>	ok	global <code>__far</code> object
<code>int __far *i;</code>	ok	pointer to a <code>__far</code> object
<code>int * __far i;</code>	ok	<code>__far</code> pointer to <code>int</code>
<code>int __far * __far i;</code>	ok	<code>__far</code> pointer to a <code>__far</code> object
<code>__far int *i;</code>	ok	pointer to a <code>__far</code> integer
<code>int * __far (* __far f)(void)</code>	ok	<code>__far</code> pointer to function returning a <code>__far</code> pointer to <code>int</code>
<code>void * __far (* f)(void)</code>	ok	pointer to function returning a <code>__far</code> pointer to <code>void</code>
<code>void __far * (* f)(void)</code>	ok	pointer to <code>__far</code> function returning a pointer to <code>void</code>

__near Keyword

NOTE	Not all Back Ends may support this keyword. See section “Non-ANSI Keywords” in the Back End.
-------------	----------------------------------------------------------------------------------------------

The `near` keyword is a synonym for `__near`. The `near` keyword is only allowed when [option -Ansi](#) is present.

The `__near` keyword can be used instead of the `__far` keyword. It is used in situations where non qualified pointers are `__far` and an explicit `__near` access should be specified or where the `__near` calling convention must be explicitly specified.

The `__near` keyword uses two semantic variations. Either it specifies a small size of a function or data pointers, or it specifies the `__near` calling convention.

Table 1.43 Interpretation of the `__near` Keyword

Declaration	Allowed	Type Description
<code>int __near f();</code>	ok	<code>__near</code> function returning an int
<code>int __near __far f();</code>	error	
<code>__near f();</code>	ok	<code>__near</code> function returning an int
<code>int __near * __far f();</code>	ok	<code>__near</code> function returning a <code>__far</code> pointer to int
<code>int __far *i;</code>	error	
<code>int * __near i;</code>	ok	<code>__far</code> pointer to int
<code>int * __far* __near i;</code>	ok	<code>__near</code> pointer to <code>__far</code> pointer to int
<code>int * __far (* __near f)(void)</code>	ok	<code>__near</code> pointer to function returning a <code>__far</code> pointer to int
<code>void * __near (* f)(void)</code>	ok	pointer to function returning a <code>__near</code> pointer to void
<code>void __far * __near (* __near f)(void)</code>	ok	<code>__near</code> pointer to <code>__far</code> function returning a <code>__far</code> pointer to void

Compatibility

`__far` pointers and normal pointers are compatible. If necessary, the normal pointer is extended to a `__far` pointer (subtraction of two pointers or assignment to a `__far` pointer). In the other case, the `__far` pointer is clipped to a normal pointer (i.e. the page part is discarded).

__alignof__ Keyword

Some processors align objects according to their type. The unary operator, **__alignof__**, determines the alignment of a specific type. By providing any type, this operator returns its alignment. This operator behaves in the same way as "sizeof(type-name)" operator. See the target backend section to check which alignment corresponds to which fundamental data type (if any is required) or to which aggregate type (structure, array).

This macro may be useful for the va_arg macro in stdarg.h, e.g. to differentiate the alignment of a structure containing four objects of four bytes from that of a structure containing two objects of eight bytes. In both cases, the size of the structure is 16 bytes but the alignment may differ, as shown:

```
#define va_arg(ap,type) \
(((__alignof__(type)>=8) ? \
((ap) = (char *)(((int)(ap) \
+ __alignof__(type) - 1) & (~(__alignof__(type) - 1)))) \
: 0), \
((ap) += __va_rounded_size(type)), \
(((type *) (ap))[-1]))
```

__va_sizeof__ Keyword

According to the ANSI-C specification, you must promote character arguments in open parameter lists to int. The use of "char" in the va_arg macro to access this parameter may not work as per the ANSI-C specification.

```
int f(int n, ...) {
    int res;
    va_list l= va_start(n, int);
    res= va_arg(l, char); /* should be va_arg(l, int) */
    va_end(l);
    return res;
}

void main(void) {
    char c=2;
    int res=f(1,c);
}
```

With the **__va_sizeof__** operator, the va_arg macro is written the way that f returns 2. A safe implementation of the f function is to use "va_arg(l, int)" instead of "va_arg(l, char)".

The `__va_sizeof__` unary operator, which is used exactly as the `sizeof` keyword, returns the size of its argument after promotion as in an open parameter list.

Examples:

```
__va_sizeof__(char) == sizeof (int)
__va_sizeof__(float) == sizeof (double)
struct A { char a; };
__va_sizeof__(struct A) >= 1 (1 if the target needs no
padding bytes)
```

NOTE It is not possible in ANSI-C to distinguish a 1-byte structure without alignment/padding from a character variable in a `va_arg` macro. They need a different space on the open parameter calls stack for some processors.

interrupt Keyword

The `_interrupt` keyword is a synonym for `interrupt`, which is allowed when [option-Ansi](#) is present.

NOTE Not all Back Ends support this keyword. See the “Non-ANSI Keywords” section in the Back End.

You can use one of two ways to specify a function to be an interrupt routine:

1. Use the [#pragma TRAP_PROC](#) and adapt the Linker parameter file.
2. USE the nonstandard interrupt keyword.

Use the nonstandard interrupt keyword like any other type qualifier. It specifies a function to be an interrupt routine. It is followed by a number specifying the entry in the interrupt vector that should contain the address of the interrupt routine. If it is not followed by any number, the interrupt keyword has the same effect as the `pragma TRAP_PROC`. It specifies a function to be an interrupt routine. Although, the number of the vector entry must be specified in the Linker parameter file with the Linker VECTOR number functionname command.

Example:

```
interrupt void f(); // ok
// same as #pragma TRAP_PROC,
// please set the entry number in prm-file
```

```
interrupt 3 int g(); // ok
    // third entry in vector points to g()
interrupt int l; // error, not a function
```

Example:

```
interrupt 2 int g();
// The 2nd entry (number 2) gets the address of func g().
```

asm Keyword

The Compiler supports target processor instructions inside of C functions.

The `__asm` keyword is a synonym for `asm`, which is allowed when [option -Ansi](#) is present.

See the inline assembler section in the backend chapter for details.

Some examples:

```
asm {
    nop
    nop ; comment
}

asm ("nop; nop");
asm( "nop\n nop");

asm "nop";
asm nop;

#asm
nop
nop
#endasm
```

Implementation-Defined Behavior

The ANSI standard contains a couple of places where the behavior of a particular Compiler is left undefined. It is possible for different Compilers to implement certain features in different ways, even if they all comply with the ANSI-C standard. Subsequently, we'll discuss those points and the behavior implemented by the Compiler.

Right Shifts

The result of $E1 >> E2$ is implementation-defined for a right shift of an object with a signed type having a negative value, if $E1$ has a signed type and a negative value.

In this implementation, an arithmetic right shift is performed.

Initialization of Aggregates with Non-Constants

The initialization of aggregates with non-constants is not allowed in the ANSI-C specification. The Compiler allows it if the option [-Ansi](#) is not set.

Example:

```
void main() {
    struct A {
        struct A *n;
    } v={&v}; /* the address of v is not constant */
}
```

Sign of char

The ANSI-C standard leaves it open, whether the data type `char` is signed or unsigned. Check the Back End chapter for data about default settings.

Division and Modulus

The results of the "/" and "%" operators are also not properly defined for signed arithmetic operations unless both operands are positive.

NOTE	The way a Compiler implements "/" and "%" for negative operands is determined by the hardware implementation of the target's division instruction(s).
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------

See [Division and Modulus](#) for description and examples.

Translation Limitations

This section describes the internal limitations of the Compiler. Some limitations are stack limitations depending on the operating system used. For example, in some operating systems, limits depend on whether the compiler is a 32-bit compiler running on a 32-bit platform (e.g. Windows NT), or if it is a 16-bit Compiler running on a 16-bit platform (e.g. Windows for Workgroups).

The ANSI column in the table below shows the recommended limitations of the ANSI C (5.2.4.1 in ISO/IEC 9899:1990 (E)) and ANSI C++ standards. These quantities are only guidelines and do not determine compliance. The ‘Implementation’ column shows the actual implementation value and the possible message number. ‘-’ means that there is no information available for this topic and ‘n/a’ denotes that this topic is not available.

Limitation	Implementation	ANSI C	ANSI C++
Nesting levels of compound statements, iteration control structures, and selection control structures	256 (C1808)	15	256
Nesting levels of conditional inclusion	-	8	256
Pointer, array, and function decorators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration	-	12	256
Nesting levels of parenthesized expressions within a full expression	32 (C4006)	32	256
Number of initial characters in an internal identifier or macro name	32767	31	1024
Number of initial characters in an external identifier	32767	6	1024
External identifiers in one translation unit	-	511	65536
Identifiers with block scope declared in one block	-	127	1024
Macro identifiers simultaneously defined in one translation unit	655'360'000 (C4403)	1024	65536
Parameters in one function definition	-	31	256
Arguments in one function call	-	31	256
Parameters in one macro definition	1024 (C4428)	31	256
Arguments in one macro invocation	2048 (C4411)	31	256
Characters in one logical source line	2^{31}	509	65536

Limitation	Implementation	ANSI C	ANSI C++
Characters in a character string literal or wide string literal (after concatenation)	8196 (C3301 , C4408 , C4421)	509	65536
Size of an object	32767	32767	262 144
Nesting levels for #include files	512 (C3000)	8	256
Case labels for a switch statement (excluding those for any nested switch statements)	1000	257	16384
Data members in a single class, structure, or union	-	127	16384
Enumeration constants in a single enumeration	-	127	4096
Levels of nested class, structure, or union definitions in a single struct declaration list	32	15	256
Functions registered by atexit()	-	n/a	32
Direct and indirect base classes	-	n/a	16384
Direct base classes for a single class	-	n/a	16384
Members declared in a single class	-	n/a	4096
Final overriding virtual functions in a class, accessible or not	-	n/a	16384
Direct and indirect virtual bases of a class	-	n/a	1024
Static members of a class	-	n/a	1024
Friend declarations in a class	-	n/a	4096
Access control declarations in a class	-	n/a	4096
Member initializers in a constructor definition	-	n/a	6144
Scope qualifications of one identifier	-	n/a	256
Nested external specifications	-	n/a	1024
Template arguments in a template declaration	-	n/a	1024

Limitation	Implementation	ANSI C	ANSI C++
Recursively nested template instantiations	-	n/a	17
Handlers per try block	-	n/a	256
Throw specifications on a single function declaration	-	n/a	256

The table below shows other limitations which are not mentioned in a ANSI standard

Limitation	Description
Type Declarations	Derived types must not contain more than 100 components.
Labels	There may be at most 16 other labels within one procedure.
Macro Expansion	Expansion of recursive macros is limited to 70 (16bit OS) or 2048 (32bit OS) recursive expansions (C4412).
Include Files	The total number of include files is limited to 8196 for a single compilation unit.
Numbers	Maximum of 655'360'000 different number for a single compilation unit (C2700 , C3302).
Goto	M68k only: Maximum of 512 gotos for a single function (C15300).
Parsing Recursion	Maximum of 1024 parsing recursions (C2803).
Lexical Tokens	Limited by memory only (C3200).
Internal ID's	Maximum of 16'777'216 internal ID's for a single compilation unit (C3304). Internal ID's are used for additional local/global variables created by the Compiler (e.g. by using CSE).
Code Size	Code size is limited to 32kB for each single function.
File Names	Maximum length for file names (including path) are 128 characters for 16 bit applications and 256 for Win32 applications. UNIX versions support file names without path of 64 characters length and 256 with path. Paths may be 96 characters on 16 bit PC versions, 192 on UNIX versions and 256 on 32 bit PC versions.

ANSI-C Standard

This section provides a short overview about the implementation (see also ANSI Standard 6.2) of the ANSI-C conversion rules.

Integral Promotions

You may use a char, a short int, or an int bitfield, or their signed or unsigned varieties, or an enum type, in an expression wherever an int or unsigned int is used. If an int represents all values of the original type, the value is converted to an int; otherwise, it is converted to an unsigned int. Integral promotions preserve value including sign.

Signed/Unsigned Integers

Promoting a signed integer type to another signed integer type of greater size requires "sign extension": In two's-complement representation, the bit pattern is unchanged, except for filling the high order bits with copies of the sign bit.

When converting a signed integer type to an unsigned inter type, if the destination has equal or greater size, the first signed extension of the signed integer type is performed. If the destination has a smaller size, the result is the remainder on division by a number, one greater than the largest unsigned number, that is represented in the type with the smaller size.

Arithmetic Conversions

The operands of binary operators do implicit conversions:

- If either operand has type long double, the other operand is converted to long double.
- If either operand has type double, the other operand is converted to double.
- If either operand has type float, the other operand is converted to float.
- The integral promotions are performed on both operands.

Then the following rules are applied:

- If either operand has type unsigned long int, the other operand is converted to unsigned long int.
- If one operand has type long int and the other has type unsigned int, if a long int can represent all values of an unsigned int, the operand of type unsigned int is converted to long int; if a long int cannot represent all the values of an unsigned int, both operands are converted to unsigned long int.
- If either operand has type long int, the other operand is converted to long int.
- If either operand has type unsigned int, the other operand is converted to unsigned int.
- Both operands have type int.

Order of Operand Evaluation

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
& / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
? :	right to left
= += -= *= /= %= &= ^= = <<= >>=	right to left
,	left to right
Unary +, - and * have higher precedence than the binary forms.	

Example:

```
if (a&3 == 2)
```

‘==’ has higher precedence than ‘&’, thus it is evaluated as

```
if (a & (3==2))
```

which is the same as

```
if (a&1)
```

Hint: use brackets if you are not sure about associativity!

Rules for Standard Type Sizes

```
sizeof(char)    <= sizeof(short)
sizeof(short)   <= sizeof(int)
sizeof(int)     <= sizeof(long)
```

```
sizeof(long)  <= sizeof(long long)
sizeof(float) <= sizeof(double)
sizeof(double)<= sizeof(long double)
```

In ANSI-C, enumerations have the type of ‘int’. In this implementation they have to be smaller or equal than ‘int’.

Floating Type Formats

The Compiler supports two IEEE floating point formats: IEEE32 and IEEE64. There may also be a DSP format supported by the processor. The figure below shows these three formats.

IEEE 32bit Format (Precision: 6.5 decimal digits)	
8bit exp	23 bit mantissa
sign bit	
$\text{value} = -1^s * 2^{(E-127)} * 1.m$	
IEEE 64bit Format (Precision: 15 decimal digits)	
11bit exp	52 bit mantissa
sign bit	
$\text{value} = -1^s * 2^{(E-1023)} * 1.m$	
DSP Format (Precision: 4.5 decimal digits)	
16bit mantissa	16bit exponent
$\text{value} = m * 2^E$ (no hidden bit)	
Negative exponents are in 2’s complement, the mantissa in signed fixed-point format.	

Floats are implemented as IEEE32, and doubles as IEEE64. This may vary for a specific Back End, or possibly, both formats may not be supported. Please check the Back End chapter for details, default settings and supported formats.

Floating Point Representation of 500.0 for IEEE

First convert 500.0 from the decimal representation to a representation with base 2:

$\text{value} = (-1)^s * m * 2^e$
where sign is 0 or 1,

$2 > m \geq 1$ for IEEE
and exp is a integral number.

For 500 this gives:

```
sign (500.0) = 1
mant (500.0,IEEE) = 1.953125
exp (500.0,IEEE) = 8
```

NOTE	The number 0 (zero) can not be represented this way. So for 0 IEEE defines a special bit pattern consisting of 0 bits only.
-------------	-----------------------------------------------------------------------------------------------------------------------------

Next convert the mantissa into its binary representation.

```
mant (500.0,IEEE) = 1.953125
= 1*2^(0) + 1*2^(-1) + 1*2^(-2) + 1*2^(-3) + 1*2^(-4)
+ 0*2^(-5) + 1*2^(-6) + 0*...
= 1.111101000... (bin).
```

Because this number is computed to be larger or equal to 1 and smaller than 2, there is always a 1 in front of the decimal point. For the remaining steps, this constant 1 is left out to save space.

```
mant (500.0, IEEE, cutted) = .111101000... .
```

The exponent must also be converted to binary format:

```
exp (500.0,IEEE) = 8 == 08 (hex) == 1000 (bin)
```

For the IEEE formats the sign is encoded as separate bit (sign magnitude representation)

Representation of 500.0 in IEEE32 Format

The exponent in IEEE32 has a fixed offset of 127 to always have positive values:

```
exp (500.0,IEEE32) = 8+127 == 87 (hex) == 10000111 (bin)
```

The fields must be put together according to the graphic:

```
500.0 (dec)
= 0 (sign) 10000111 (exponent)
    111101000000000000000000 (mantissa) (IEEE32 as bin)
= 0100 0011 1111 1010 0000 0000 0000 0000 (IEEE32 as bin)
= 43 fa 00 00 (IEEE32 as hex)

-500.0 (dec)
= 1 (sign) 10000111 (exponent)
    111101000000000000000000 (mantissa) (IEEE32 as bin)
```

```
= 1100 0011 1111 1010 0000 0000 0000 0000 (IEEE32 as bin)
= C3 fa 00 00 (IEEE32 as hex)
```

Representation of 500.0 in IEEE64 Format

The exponent in IEEE64 has a fixed offset of 1023 to always have positive values:

```
exp (500.0, IEEE64) = 8+1023 == 407 (hex) == 10000000111  
(bin)
```

The IEEE64 format is similar to IEEE32 except that more bits are used to represent the exponent and the mantissa.

NOTE The IEEE formats recognize several special bit patterns for special values. The number 0 (zero) is encoded by the bit pattern consisting of zero bits only. Other special values such as “Not a number”, “infinity”, -0 (minus zero) and denormalized numbers do exist. Please refer to the IEEE standard documentation for details. Except for the 0 (zero) and -0 (minus zero) special formats, not all special formats may be supported for specific backends.

Representation of 500.0 in DSP Format

Convert 500.0 from the decimal representation to a representation with base 2. In contradiction to IEEE, DSP normalizes the mantissa between 0 and 1 and not between

1 and 2. This makes it possible to also represent 0, which must have a special pattern in IEEE. Also, the exponent is different from IEEE.

```
value = (-1)^s * m*2^e
where sign is 1 or -1,
1 > m >= 0
and exp is a integral number.
```

For 500 this gives:

```
sign (500.0) = 1
mant (500.0,DSP) = 0.9765625
exp (500.0,DSP) = 9
```

Next convert the mantissa into its binary representation.

```
mant (500.0, DSP) = 0.9765625 (dec)
= 0*2^(0) + 1*2^(-1) + 1*2^(-2) + 1*2^(-3) + 1*2^(-4) +
1*2^(-5)
+ 0*2^(-6) + 1*2^(-7) + 0*...
= 0.1111101000... (bin).
```

Because this number is computed to be always larger or equal to 0 and smaller than 1, there is always a 0 in front of the decimal point. For the remaining steps this constant is left out to save space. There is always a 1 after the decimal point, except for 0 and intermediate results. This bit is encoded, so the DSP loses one additional bit of precision compared with IEEE.

```
mant (500.0, DSP, cutted) = .1111101000... .
```

The exponent must also be converted to binary format:

```
exp (500.0,DSP) = 9 == 09 (hex) == 1001 (bin)
```

Negative exponents are encoded by the 2's representation of the positive value.

The sign is encoded into the mantissa by taking the 2's complement for negative numbers and adding a 1 bit in the front. For DSP and positive numbers a 0 bit is added at the front.

```
mant(500.0, DSP) = 0111110100000000 (bin)
```

The two's complement is taken for negative numbers:

```
mant(-500.0, DSP) = 1000001100000000 (bin)
```

Finally the mantissa and the exponent must be joined according to the graphic above:

```
500.0 (dec)
= 7D 00 (mantissa) 00 09 (exponent) (DSP as hex)
= 7D 00 00 09 (DSP as hex)
= 0111 1101 0000 0000 0000 0000 1001 (DSP as bin)
```

```
-500.0 (dec)
= 83 00 (mantissa) 00 09 (exponent) (DSP as hex)
= 83 00 00 09 (DSP as hex)
= 1000 0011 0000 0000 0000 0000 0000 1001 (DSP as bin)
```

NOTE	The order of the byte representation of a floating point value depends on the byte ordering of the backend. The first byte in the previous diagrams must be considered as the most significant byte.
-------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Volatile Objects, Absolute Variables

The Compiler does not do register- and constant tracing on volatile/absolute global objects. Accesses to volatile/absolute global objects are not eliminated.

Example:

```
volatile int x;

void main(void) {
    x = 0;
    ...
    if(x == 0) { // without volatile attribute, the
                  // comparison may be optimized away!
        Error(); // Error() is called without compare!
    }
}
```

Bit Fields

There is no standard way to allocate bit fields. Bit-field allocation varies from Compiler to Compiler even for the same target. Using bit fields for access to I/O registers is non-portable and inefficient for the masking involved in unpacking individual fields. It is recommended that you use regular bit-and (&) and bit-or (|) operations for I/O port access.

The maximum width of bit fields is Back End dependent (see Back End for details), in that plain `int` bit fields are signed. A bit field never crosses a word (2 bytes) boundary. As stated in Kernighan and Ritchie's "*The C Programming Language*", 2ND ed., the use of bit fields is equivalent to using bit masks to which the operators &, |, ~, |= or &= are applied. In fact, the Compiler translates bit field operations to bit mask operations.

Signed Bit Fields

A common mistake is to use signed bit fields, but testing them as if they were unsigned. Signed bit fields have a value of -1 or 0. Consider the following example:

```
typedef struct _B {
    signed int b0: 1;
} B;

B b;

if (b.b0 == 1) ...
```

The Compiler issues a warning and replaces the 1 with -1 because the condition `(b.b0 == 1)` does not make sense, i.e. it is always false. The test `(b.b0 == -1)` is performed as expected. This substitution is not ANSI compatible and will not be performed when the [-Ansi](#) Compiler option is active.

The correct way to specify this is with an unsigned bit field. Unsigned bit fields have the values 0 or 1.

```
typedef struct _B {
    unsigned b0: 1;
} B;

B b;

if (b.b0 == 1) ...
```

Because `b0` is an unsigned bit field having the values 0 or 1, the test `(b.b0 == 1)` is correct.

Recommendations

In order to save memory, it recommended to implement globally accessible boolean flags as unsigned bit fields of width 1. However, it is not recommend using bit fields for other purposes because:

- Using bit fields to describe a bit pattern in memory is not portable between Compilers, even on the same target, as different Compilers may allocate bit fields differently.

For information about how the Compiler allocates bit fields, see the *Back End, Data Types* section.

Segmentation

The Linker supports the concept of segments in that the memory space may be partitioned into several segments. The Compiler allows attributing a certain segment

name to certain global variables or functions which then will be allocated into that segment by the Linker. Where that segment actually lies is determined by an entry in the Linker parameter file.

The syntax of the segment specification pragma:

```
SegDef= "#pragma" SegmentType ( {SegmentMod} SegmentName | "DEFAULT" ).  
  
SegmentType= "CODE_SEG" | "CODE_SECTION" |  
             "DATA_SEG" | "DATA_SECTION" |  
             "CONST_SEG" | "CONST_SECTION" |  
             "STRING_SEG" | "STRING_SECTION".  
  
SegmentMod= "__DIRECT_SEG" | "__NEAR_SEG" | "__CODE_SEG"  
          | "__FAR_SEG" | "__BIT_SEG" | "__Y_BASED_SEG"  
          | "__Z_BASED_SEG" | "__DPAGE_SEG" | "__PPAGE_SEG"  
          | "__EPAGE_SEG" | "__RPAGE_SEG" | "__GPAGE_SEG"  
          | "__PIC_SEG" | CompatSegmentMod.  
  
CompatSegmentMod= "DIRECT" | "NEAR" | "CODE" | "FAR" | "BIT"  
                  | "Y_BASED" | "Z_BASED" | "DPAGE" | "PPAGE"  
                  | "EPAGE" | "RPAGE" | "GPAGE" | "PIC".
```

Since there are two basic types of segments, code and data segments, there are also two pragmas to specify segments:

```
#pragma CODE_SEG <segment_name>  
#pragma DATA_SEG <segment_name>
```

In addition there are pragmas for constant data and for strings:

```
#pragma CONST_SEG <segment_name>  
#pragma STRING_SEG <segment_name>
```

All four pragmas are valid until the next pragma of the same kind is encountered.

In the HIWARE object file format, constants are put into the DATA_SEG if no CONST_SEG was specified. In the ELF Object file format, constants are always put into a constant segment.

Strings are put into the segment STRINGS until a pragma STRING_SEG is specified. After this pragma, all strings are allocated into this constant segment. The linker then treats this segment like any other constant segment.

If no segment is specified, the Compiler assumes two default segments named DEFAULT_ROM (the default code segment) and DEFAULT_RAM (the default data segment). Use the segment name DEFAULT to explicitly make these default segments the current segments :

```
#pragma CODE_SEG DEFAULT
#pragma DATA_SEG DEFAULT
#pragma CONST_SEG DEFAULT
#pragma STRING_SEG DEFAULT
```

Segments may also be declared as `__SHORT_SEG` by inserting the keyword `__SHORT_SEG` just before the segment name (with the exception of the predefined segment `DEFAULT` – this segment cannot be qualified with `__SHORT_SEG`). This makes the Compiler use short (i.e. 8 bits or 16 bits, depending on the Back End) absolute addresses to access global objects, or to call functions. It is the programmer's responsibility to allocate `__SHORT_SEG` segments in the proper memory area.

NOTE The default code and data segments may not be declared as `__SHORT_SEG`.

The meaning of the other segment modifiers, such as `__NEAR_SEG` and `__FAR_SEG`, are backend specific. Modifiers that are not supported by the back end are ignored. Please refer to the backend chapter for data about which modifiers are supported.

The segment pragmas also have an effect on static local variables. Static local variables are local variables with the ‘static’ flag set. They are in fact normal global variables but with scope only to the function in which they are defined:

```
#pragma DATA_SEG MySeg
static char foo(void) {
    static char i = 0; /* place this variable into MySeg */
    return i++;
}
#pragma DATA_SEG DEFAULT
```

NOTE Using the ELF/DWARF object file format (option [-F1](#) or [-F2](#)), all constants are placed into the section `.rodata` by default unless a `#pragma CONST_SEG` is used.

NOTE There are aliases to satisfy the ELF naming convention for all segment names. Use `CODE_SECTION` instead of `CODE_SEG`. Use `DATA_SECTION` instead of `DATA_SEG`. Use `CONST_SECTION` instead of `CONST_SEG`. Use `STRING_SECTION` instead of

STRING_SEG. These aliases behave exactly like the XXX_SEG name versions.

Example of Segmentation Without Compiler -Cc Option

```
/* Placed into Segment: */
static int a;                                /* DEFAULT_RAM(-1) */
static const int c0 = 10;                      /* DEFAULT_RAM(-1) */

#pragma DATA_SEG MyVarSeg
static int b;                                /* MyVarSeg(0) */
static const int c1 = 11;                      /* MyVarSeg(0) */

#pragma DATA_SEG DEFAULT
static int c;                                /* DEFAULT_RAM(-1) */
static const int c2 = 12;                      /* DEFAULT_RAM(-1) */

#pragma DATA_SEG MyVarSeg
#pragma CONST_SEG MyConstSeg
static int d;                                /* MyVarSeg(0) */
static const int c3 = 13;                      /* MyConstSeg(1) */

#pragma DATA_SEG DEFAULT
static int e;                                /* DEFAULT_RAM(-1) */
static const int c4 = 14;                      /* MyConstSeg(1) */

#pragma CONST_SEG DEFAULT
static int f;                                /* DEFAULT_RAM(-1) */
static const int c5 = 15;                      /* DEFAULT_RAM(-1) */
```

Example for Segmentation With Compiler -Cc Option

```
Placed into Segment:
static int a;                                /* DEFAULT_RAM(-1) */
static const int c0 = 10;                      /* ROM_VAR(-2) */

#pragma DATA_SEG MyVarSeg
static int b;                                /* MyVarSeg(0) */
static const int c1 = 11;                      /* MyVarSeg(0) */

#pragma DATA_SEG DEFAULT
static int c;                                /* DEFAULT_RAM(-1) */
```

```
static const int c2 = 12;          /* ROM_VAR(-2) */

#pragma DATA_SEG MyVarSeg
#pragma CONST_SEG MyConstSeg
static int d;                    /* MyVarSeg(0) */
static const int c3 = 13;          /* MyConstSeg(1) */

#pragma DATA_SEG DEFAULT
static int e;                    /* DEFAULT_RAM(-1) */
static const int c4 = 14;          /* MyConstSeg(1) */

#pragma CONST_SEG DEFAULT
static int f;                    /* DEFAULT_RAM(-1) */
static const int c5 = 15;          /* ROM_VAR(-2) */
```

Optimizations

The Compiler applies a variety of code improving techniques under the term "optimization". This section provides a short overview about the most important optimizations.

Peephole Optimizer

A peephole optimizer is a simple optimizer in a Compiler. A peephole optimizer tries to optimize specific code patterns on speed or code size. After recognizing these specific patterns, they will be replaced by other optimized patterns.

After code is generated by the back end of an optimizing Compiler, it is still possible that code patterns may result that are still capable of being optimized. The optimizations of the peephole optimizer are highly back end dependent because the peephole optimizer was implemented with characteristic code patterns of the back end in mind.

Certain peephole optimizations only make sense in conjunction with other optimizations, or together with some code patterns. These patterns may have been generated by doing other optimizations. There are optimizations (e.g removing of a branch to the next instructions) that are removed by the peephole optimizer, though they could have been removed by the branch optimizer as well. Such simple branch optimizations are performed in the peephole optimizer to reach new optimizable states.

Strength Reduction

Strength reduction is an optimization that strives to replace expensive operations by cheaper ones, where the cost factor is either execution time or code size. Examples are the replacement of multiplication and division by constant powers of two with left or right shifts.

NOTE	The compiler can only replace a division by two using a shift operation if either the target division is implemented the way that $-1/2 == -1$, or if the value to be divided is unsigned. The result is different for negative values. To give the compiler the possibility to use a shift, the C source code should already contain a shift, or the value to be shifted should be unsigned.
-------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Shift Optimizations

Shifting a byte variable by a constant number of bits is intensively analyzed. The Compiler always tries to implement such shifts in the most efficient way.

Branch Optimizations

This optimization tries to minimize the span of branch instructions. The Compiler will never generate a long branch where a short branch would have sufficed. Also, branches to branches may be resolved into two branches to the same target. Redundant branches (e.g. a branch to the instruction immediately following it) may be removed.

Dead Code Elimination

The Compiler removes dead assignments while generating code. In some programs it may find additional cases of expressions that are not used.

Constant Variable Optimization

If a constant non-volatile variable is used in any expression, the Compiler replaces it by the constant value it holds. This needs less code than taking the object itself.

The constant non-volatile object itself is removed if there is no expression taking the address of it. This results in using less memory space.

Example:

```
void f(void) {
    const int ci = 100; // ci removed (no address taken)
    const int ci2 = 200; // ci2 not removed
                        // (address taken below)
    const volatile int ci3 = 300; //ci3 not removed (volatile)
    int i;
    int *p;
    i = ci;      // replaced by i=100;
    i = ci2;     // no replacement
    p = &ci2;    // address taken
}
```

Global constant non-volatile variables are not removed. Their use in expressions are replaced by the constant value they hold.

Constant non-volatile arrays are also optimized.

Example:

```
void g(void) {
    const int array[] = {1,2,3,4};
    int i;
    i = array[2]; // replaced by i=3;
}
```

Tree Rewriting

The structure of the intermediate code between Front End and Back End allows the Compiler to perform some optimizations on a higher level. Examples are shown in the following sections.

Switch Statements

Efficient translation of switch statements is mandatory for any C Compiler. The Compiler applies different strategies, i.e. branch trees, jump tables, and a mixed strategy, depending on the case label values and their numbers.

Method	Description
Branch Sequence	For small switches with scattered case label values, the Compiler generates an if ... elseif ... elseif ... else ... sequence if the Compiler switch -Os is active.
Branch Tree	For small switches with scattered case label values, the Compiler generates a branch tree. This is the equivalent to unrolling a binary search loop of a sorted jump table and therefore is very fast. However, there is a point at which this method is not feasible simply because it uses too much memory.
Jump Table	In such cases, the Compiler creates a table plus a call of a switch processor. There are two different switch processors. If there are a lot of labels with more or less consecutive values, a direct jump table is used. If the label values are scattered, a binary search table is used.
Mixed Strategy	Finally, there may be switches having "clusters" of label values separated by other labels with scattered values. In this case, a mixed strategy is applied, generating branch trees or search tables for the scattered labels and direct jump tables for the clusters.

Absolute Values

Another example for optimization on a higher level is the calculation of absolute values. In C, the programmer has to write something on the order of:

```
float x, y;  
x = (y < 0.0) ? -y : y;
```

This results in lengthy and inefficient code. The Compiler recognizes cases like this and treats them specially in order to generate the most efficient code. Only the most significant bit has to be cleared.

Combined Assignments

The Compiler can also recognize the equivalence between the three following statements:

```
x = x + 1;  
x += 1;  
x++;
```

and between:

```
x = x / y;  
x /= y;
```

Therefore, the Compiler generates equally efficient code for either case.

Using Qualifiers for Pointers

The use of qualifiers (const, volatile, ...) for pointers is confusing. This section provides some examples for the use of const/volatile as const and volatile are very common for Embedded Programming.

Consider the following example:

```
int i;  
const int ci;
```

The above definitions are: a ‘normal’ variable ‘i’ and a constant variable ‘ci’. Each are placed into ROM. Note that for C++, the constant ‘ci’ must be initialized.

```
int *ip;  
const int *cip;
```

‘ip’ is a pointer to an ‘int’, where ‘cip’ is a pointer to a ‘const int’.

```
int *const icp;  
const int *const cicp;
```

‘icp’ is a ‘const pointer’ to an ‘int’, where ‘cicp’ is a ‘const pointer’ to a ‘const int’.

It helps if you know that the qualifier for such pointers is always on the right side of the ‘*’. Another way is to read the source from right to left.

You can express this rule in the same way to volatile. Consider the following example of an ‘array of five constant pointers to volatile integers’:

```
volatile int *const arr[5];
```

‘arr’ is an array of five constant pointers pointing to volatile integers. Because the array itself is constant, it is put into ROM. It does not matter if the array is constant or not regarding where the pointers point to. Consider the next example?

```
const char *const *buf[] = {&a, &b};
```

Because the array of pointers is initialized, the array is not constant. ‘buf’ is a (non-constant) array of two pointers to constant pointers which points to constant characters. Thus ‘buf’ cannot be placed into ROM by the Compiler/Linker.

Consider a constant array of five ordinary function pointers. Assuming that:

```
void (*fp)(void);
```

is a function pointer ‘fp’ returning void and having void as parameter, you can define it with:

```
void (*fparr[5])(void);
```

It is also possible to use a typedef to separate the function pointer type and the array:

```
typedef void (*Func)(void);
Func fp;
Func fparr[5];
```

You can write a constant function pointer as:

```
void (*const cfp) (void);
```

Consider a constant function pointer having a constant int pointer as a parameter returning void:

```
void (*const cfp2) (int *const);
```

Or a const function pointer returning a pointer to a volatile double having two constant integers as parameter:

```
volatile double *(*const fp3) (const int, const int);
```

And an additional one:

```
void (*const fp[3])(void);
```

This is an array of three constant function pointers, having void as parameter and returning void. ‘fp’ is allocated in ROM because the ‘fp’ array is constant.

Consider an example using function pointers:

```
int (* (** func0(int (*f) (void)))) (int (*) (void)) (int
(*) (void)) {
    return 0;
}
```

It is actually a function called func. This func has one function pointer argument called f. The return value is more complicated in this example. It is actually a function pointer of a complex type. Here we do not explain where to put a const so that the destination of the returned pointer cannot be modified. Alternately, the same function is written more simply using typedefs:

```
typedef int (*funcType1) (void);
typedef int (* funcType2) (funcType1);
typedef funcType2 (* funcType3) (funcType1);
```

```
funcType3* func0(funcType1 f) {
    return 0;
}
```

Now, the places of the const becomes obvious. Just behind the * in funcType3:

```
typedef funcType2 (* const constFuncType3) (funcType1);
constFuncType3* func1(funcType1 f) {
    return 0;
}
```

By the way, also in the first version here is the place where to put the const:

```
int (* (*const * func1(int (*f) (void))) (int (*) (void)))
                                         (int (*) (void)) {
    return 0;
}
```

Defining C Macros Containing HLI Assembler Code

You can define some ANSI C macros that contain HLI assembler statements when you are working with the HLI assembler. Because HLI assembler is heavily Back End dependent, the following example uses a pseudo Assembler Language:

```
CLR Reg0      ; Clear Register zero
CLR Reg1      ; Clear Register one
CLR var       ; Clear variable 'var' in memory
LOAD var,Reg0 ; Load the variable 'var' into Register 0
LOAD #0, Reg0 ; Load immediate value zero into Register 0
LOAD @var,Reg1 ; Load address of variable 'var' into Reg1
STORE Reg0,var ; Store Register 0 into variable 'var'
```

The HLI instructions are only used as a possible example. For real applications, you must replace the above pseudo HLI instructions with the HLI instructions for your target.

Defining a Macro

An HLI assembler macro is defined using the preprocessor directive ‘define’.

Example:

```
/* Following macro clears the register zero. */
#define ClearReg0 {asm CLR Reg0;}
```

The macro is invoked in the following way in the source code:

```
ClearReg0;
```

The preprocessor expands the macro:

```
{asm CLR Reg0;};
```

An HLI assembler macro can contain one or several HLI assembler instructions. As the ANSI C preprocessor expands a macro on a single line, you cannot define an HLI assembler block in a macro. You can, however, define a list of HLI assembler instructions.

Example:

```
/* Following macro clears the registers 0 and 1. */
#define ClearReg0And1 {asm CLR Reg0; asm CLR Reg1}
```

The macro is invoked in the following way in the source code:

```
ClearReg0And1;
```

The preprocessor expands the macro:

```
{asm CLR Reg0; asm CLR Reg1;};
```

You can define an HLI assembler macro on several lines using the line separator ‘\’.

NOTE This may enhance the readability of your source file. However, the ANSI C preprocessor still expands the macro on a single line.

Example:

```
/* Following macro clears the registers 0 and 1. */
#define ClearR0andR1 {asm CLR Reg0; \
                     asm CLR Reg1}
```

The macro is invoked in the following way in the source code:

```
ClearR0andR1;
```

The preprocessor expands the macro:

```
{asm CLR Reg0; asm CLR Reg1;};
```

Using Macro Parameters

An HLI assembler macro may have some parameters which are referenced in the macro code.

Example:

```
/* This macro initializes the specified variable to 0.*/
#define Clear(var) {asm CLR var}
```

The macro is invoked in the following way in the source code:

```
Clear(var1);
```

The preprocessor expands the macro:

```
{asm CLR var1 ; };
```

Using Immediate Addressing Mode in HLI Assembler Macros

There may be one ambiguity if you are using the immediate addressing mode inside of a macro.

For the ANSI C preprocessor, the symbol # inside of a macro has a specific meaning (string constructor).

Using the [`#pragma NO_STRING_CONSTR`](#), you can tell the Compiler that in all the macros defined afterward, instructions should remain unchanged where ever the symbol # is specified. This macro is valid for the rest of the file it is specified in.

Example:

```
/* This macro initializes the specified variable to 0.*/
#pragma NO_STRING_CONSTR
#define Clear(var) {asm LOAD #0,Reg0; asm STORE Reg0,var}
```

The macro is invoked in the following way in the source code:

```
Clear(var1);
```

The preprocessor expands the macro:

```
{ asm LOAD #0,Reg0; asm STORE Reg0,var1 };
```

Generating unique Labels in HLI Assembler Macros

When some labels are defined in HLI Assembler Macros, if you invoke the same macro twice in the same function, the ANSI C preprocessor generates the same label twice (once in each macro expansion). Use the special string concatenation operator of the ANSI C preprocessor ('##') in order to generate unique labels. See [Listing 1.19](#).

Listing 1.19 Using the ANSI C Preprocessor String Concatenation Operator

```
/* The following macro copies the string pointed to by 'src'
   into the string pointed to by 'dest'.
   'src' and 'dest' must be valid array of characters.
   'inst' is the instance number of the macro call. This
   parameter must different for each invocation of the
   macro to allow the generation of unique labels. */
#pragma NO_STRING_CONSTR
#define copyMacro2(src, dest, inst) { \
    asm           LOAD @src,Reg0; /* load src addr */ \
    asm           LOAD @dest,Reg1; /* load dst addr */ \
    asm           CLR Reg2;      /* clear index reg */ \
    asm lp##inst: LOADB (Reg2, Reg0); /* load byte reg indir */ \
    asm           STOREB (Reg2, Reg1); /* store byte reg indir */ \
    asm           ADD #1,Reg2; /* increment index register */ \
    asm           TST Reg2; /* test if not zero */ \
    asm           BNE lp##inst; }
```

The `copyMacro2` macro is invoked in the following way in the source code:

```
copyMacro2(source2, destination2, 1);
copyMacro2(source2, destination3, 2);
```

During expansion of the first macro, the preprocessor generates an 'lp1' label. During expansion of the second macro, an 'lp2' label is created.

Generating Assembler Include Files (Option [-La](#))

In many projects it often make sense to use both a C compiler and an assembler. Both have different advantages. The compiler uses portable and readable code while the assembler provides full control for time-critical applications, or for direct accessing of the hardware.

The compiler can not read include files of the assembler as the assembler can not read the header files of the compiler.

The assembler include file output of the compiler lets both tools using one single source to share constants, variable/labels, and even structure fields.

The compiler writes an output file in the format of the assembler which contains all needed information of a C header file.

The current implementation supports the following mappings:

- Macros
C defines are translated to assembler EQU directives.
- enum values
C enum values are translated to EQU directives.
- C types
The size of any type and the offset of structure fields is emitted for all typedefs.
For bitfield structure fields, the bit offset and the bit size are also emitted.
- Functions
For each function a XREF entry is emitted.
- Variables
C Variables are emitted with a XREF. For structures or unions additionally all fields are defined with a EQU directive.
- Comments
C style comments /* ... */ are included as assembler comments (;....).

General

A header file must be specially prepared to emit the assembler include file.

The assembler output is enabled with a pragma anywhere in the header file:

```
#pragma CREATE_ASM_LISTING ON
```

Only macro definitions and declarations behind this pragma are emitted. The compiler stops emitting future elements when the pragma [CREATE_ASM_LISTING](#) occurs with an OFF parameter.

```
#pragma CREATE_ASM_LISTING OFF
```

Not all entries generate legal assembler constructs. Care must be taken for macros. The compiler does not check for legal assembler syntax when translating macros. Macros containing elements not supported by the assembler, should be in a section controlled by a "#pragma [CREATE_ASM_LISTING](#) OFF".

The compiler only creates an output file when the -la option is specified, and the compiled sources contain a #pragma [CREATE_ASM_LISTING](#) ON.

Example:

Header file: a.h

```
#pragma CREATE_ASM_LISTING ON
```

```
typedef struct {
    short i;
    short j;
} Struct;

Struct Var;

void f(void);

#pragma CREATE_ASM_LISTING OFF
```

When the compiler reads this header file with the option [-La](#)=a.inc a.h, it generates the following a.inc file:

Struct_SIZE	EQU \$4
Struct_i	EQU \$0
Struct_j	EQU \$2
	XREF Var
Var_i	EQU Var + \$0
Var_j	EQU Var + \$2
	XREF f

You can now use the assembler INCLUDE directive to include this file into any assembler file. The content of the C variable, Var_i, can also be accessed from the assembler without any uncertain assumptions about the alignment used by the compiler. Also, whenever a field is added to the structure Struct, the assembler code must not be altered. You must, however, regenerate the a.inc file with a make tool.

Usually the assembler include file is not created every time the compiler reads the header file. It is created only a separate pass when the header file has changed significantly. The -la option is only specified when the compiler must generate a.inc. If -la is always present, a.inc is always generated. A make tool will always restart the assembler as the assembler files depend on a.inc. Such a makefile might be similar to:

```
a.inc : a.h
$(CC) -la=a.inc a.h

a_c.o : a_c.c a.h
$(CC) a_c.c

a_asm.o : a_asm.asm a.inc
$(ASM) a_asm.asm
```

The order of elements in the header file is the same as the order of the elements in the created file, except that comments may be inside of elements in the C file. In this case, the comments may be before or after the whole element.

The order of defines does not matter for the compiler. The order of EQU directives does matter for the assembler. If the assembler has problems with the EQU directives order in a generated file, the corresponding header file must be changed accordingly.

Macros

The translation of defines is done lexically and not semantically. So the compiler does not check the accuracy of the define.

The following example shows some legal uses of this feature:

```
#pragma CREATE_ASM_LISTING ON
int i;
#define UseI i
#define Constant 1
#define Sum Constant+0X1000+01234
```

Creates:

	XREF i
UseI	EQU i
Constant	EQU 1
Sum	EQU Constant + \$1000 + @234

The hexadecimal C constant 0x1000 was translated to \$1000 while the octal 01234 was translated to @1234. Additionally, the compiler has inserted one space between every two tokens. These are the only changes the compiler make in the assembler listing for defines.

Macros with parameters, predefined macros, and macros with no defined value are not emitted.

The following defines do not work or are not emitted:

```
#pragma CREATE_ASM_LISTING ON
int i;
#define AddressOfI &i
#define ConstantInt ((int)1)
#define Mul7(a) a*7
#define Nothing
#define useUndef UndefFkt*6
#define Anything $ $ / % & % / & + * % c 65467568756 86
```

Creates:

	XREF i
AddressOfI	EQU & i
ConstantInt	EQU ((int) 1)
useUndef	EQU UndefFkt * 6
Anything	EQU \$ \$ / % & % / & + * % c
65467568756 86	

The AddressOfI macro does not assemble because the assembler does not know the C address operator &. Also, other C-specific operators such as dereferenciation (*ptr)

must not be used. The compiler emits them into the assembler listing file without any translation.

The ConstantInt macro does not work because the assembler does not know the cast syntax and the types.

Macros with parameters are not written to the listing,. Therefore, Mul7 does not occur in the listing. Also, macros just defined with no actual value as Nothing are not emitted.

The C preprocessor does not care about the syntactical content of the macro, though the assembler EQU directive does. Therefore, the compiler has no problems with the useUndef macro using the undefined object UndefFkt. The assembler EQU directive requires that all used objects are defined.

The Anything macro shows that the compiler does not care about the content of a macro. The assembler, of course, cannot treat these random characters.

These types of macros are in a header file used to generate the assembler include file. They must only be in a region started with a "#pragma [CREATE_ASM_LISTING OFF](#)" so that the compiler will not emit anything for them.

Enums

Enums in C have a unique name and a defined value. They are simply emitted by the compiler as an EQU directive.

```
#pragma CREATE_ASM_LISTING ON
enum {
    E1=4,
    E2=47,
    E3=-1*7
};
```

Creates:

E1	EQU \$4
E2	EQU \$2F
E3	EQU \$FFFFFF9

NOTE Negative values are emitted as 32-bit hex numbers.

Types

As it does not make sense to emit the size of any occurring type, only typedefs are considered.

The size of the newly defined type is specified for all typedefs. As name, the typedef name with an additional end "_SIZE" is used. For structures, the offset of all structure fields relative to the structure start is emitted. The name of the structure offsets is built by using the typedef name, and then the structure fields name after an underline ("_").

```
#pragma CREATE_ASM_LISTING ON

typedef long LONG;
struct tagA {
    char a;
    short b;
};

typedef struct {
    long d;
    struct tagA e;
    int f:2;
    int g:1;
} str;
```

Creates:

LONG_SIZE	EQU \$4
str_SIZE	EQU \$8
str_d	EQU \$0
str_e	EQU \$4
str_e_a	EQU \$4
str_e_b	EQU \$5
str_f	EQU \$7
str_f_BIT_WIDTH	EQU \$2
str_f_BIT_OFFSET	EQU \$0
str_g	EQU \$7
str_g_BIT_WIDTH	EQU \$1
str_g_BIT_OFFSET	EQU \$2

All structure fields inside of structures are contained. The generated name contains all names of all files listed in the path. If any element of the path does not have a name (e.g. an anonymous union), this element is not emitted.

The width and the offset are additionally emitted for all bitfield members. The offset 0 specifies the least significant bit, which is accessed with mask 0x1. The offset 2 the most significant bit, which is accessed with mask 0x4. The width specifies the number of bits.

The offsets, bit widths and bit offsets, given here are examples. Different compilers may emit different values. In C, the structure alignment and the bitfield allocation is determined by the compiler which specifies the correct values.

Functions

Declared functions are emitted by the XREF directive. This enables them to be used with the assembler. The function to be called from C, but defined in assembler, should not be emitted into the output file as the assembler does not allow the redefinition of labels declared with XREF. Such function prototypes are placed in an area started with a "#pragma [CREATE_ASM_LISTING OFF](#)", as shown below.

```
#pragma CREATE_ASM_LISTING ON
void main(void);
void f_C(int i, long l);

#pragma CREATE_ASM_LISTING OFF
void f_asm(void);
```

Creates:

```
XREF main
XREF f_C
```

Variables

Variables are declared with XREF. Additionally for structures, every field is defined with an EQU directive. For bitfields, the bit offset and bit size are also defined.

Variables in the __SHORT_SEG segment are defined with XREF.B to inform the assembler about the direct access. Fields in structures in __SHORT_SEG segments, are defined with a EQU.B directive.

```
#pragma CREATE_ASM_LISTING ON
struct A {
    char a;
    int i:2;
};
struct A VarA;
#pragma DATA_SEG __SHORT_SEG ShortSeg
int VarInt;
```

Creates:

VarA_a	XREF VarA
VarA_i	EQU VarA + \$0
VarA_i_BIT_WIDTH	EQU VarA + \$1
VarA_i_BIT_OFFSET	EQU \$2
	EQU \$0
	XREF.B VarInt

The variable size is not explicitly written. To emit the variable size, use a typedef with the variable type.

The offsets, bit widths and bit offsets, given here are examples. Different compilers may emit different values. In C, the structure alignment and the bitfield allocation is determined by the compiler which specifies the correct values.

Comments

Comments inside a region emitted with "#pragma [CREATE_ASM_LISTING](#) ON" are also written on a single line in the assembler include file.

Comments inside of a typedef, a structure, or a variable declaration, are placed either before or after the declaration. They are never placed inside the declaration, even if the declaration contains multiple lines. Therefore, a comment after a structure field in a typedef, is written before or after the whole typedef, not just after the type field. Every comment is on a single line. An empty comment /* */ inserts an empty line into the created file.

Example:

```
#pragma CREATE_ASM_LISTING ON
/*
   The function main is called by the startup code.
   The function is written in C. It is responsible
   to initialize the application. */
void main(void);
/*
   The SIZEOF_INT macro specified the size of an integer type
   in the compiler. */
typedef int SIZEOF_INT;
#pragma CREATE_ASM_LISTING OFF
```

Creates:

```
;;
;   The function main is called by the startup code.
;   The function is written in C. It is responsible
;   to initialize the application.
;                           XREF main
;;
;   The SIZEOF_INT macro specified the size of an integer
;   type
;   in the compiler.
SIZEOF_INT_SIZE      EQU $2
```

Guidelines

The `-La` option translates specified parts of header files into an include file to import labels and defines into an assembler source. Because the `-La` option is a very powerful option, incorrect use must be avoided using the following guidelines implemented in a real project. This section describes how the programmer uses this option to combine C and assembler sources, both using common header files.

The following recommendations help to avoid problems when writing software using the common header file technique.

General Implementation Guide Lines

- All interface memory reservations/definitions must be made in C source files. Memory areas, only accessed from assembler files, can still be defined in the common assembler way.
- Compile only C header files (and not the C source files) with the `-La` option to avoid multiple defines and other problems. The project-related makefile must contain an inference rules section that defines the C header file(s)-dependent include files to be created.
- Use `#pragma CREATE_ASM_LISTING ON/OFF` only in C header files. This `#pragma` selects the objects which should be translated to the assembler include file. The created assembler include file then holds the corresponding assembler directives.
- The `-La` option should not be part of the command line options used for all compilations. Use this option in combination with the option `-Cx` (no Code Generation). Without this option, the compiler creates an object file which could accidentally overwrite a C source object file.
- Remember to extend the list of dependencies for assembler sources in your make file.
- Check if the compiler-created assembler include file is included into your assembler source.

Notes

- In case of a zero page declared object (if this is supported by the target), the compiler translates it into an XREF.B directive for the base address of a variable/constant. The compiler translates structure fields in the zero page into a EQU.B directive in order to access them. Explicit zero page addressing syntax may be necessary as some assemblers use extended addresses to EQU.B defined labels.
- Project-defined data types must be declared in the C header file by including a global project header (e.g. `global.h`). This is necessary as the header file is compiled in a stand-alone fashion.

C++ Front End

C++

Overview

This chapter provides an overview of the implementation of the C++ language used in this Compiler. It explains the main features of the language together with information about their implementation, taking into account the differences between various Compiler vendors.

The impacts of C++ for embedded applications are also discussed.

Implementation

The C++ Compiler described in this section is based on the book "Ellis/Stroustrup: The Annotated C++ Reference Manual".

Programmers not familiar with C++ should read the following additional documentation:

- "Ellis/Stroustrup: The Annotated C++ Reference Manual", ANSI Base Document, 1995, Addison-Wesley, ISBN 0-201-51459-1
- "Bjarne Stroustrup: The C++ Programming Language", 1987, Addison-Wesley, ISBN 0-201-12078-X
- "Bjarne Stroustrup: The Design and Evolution of C++", 1994, ISBN 0-201-54330-3
- "P.J. Plauger: The Draft Standard C++ Library", 1995, Prentice Hall, ISBN 0-13-117003-1

Features

The following C++ features are supported by the cC++ Compiler:

- Stronger type checking
- C++ standard name encoding
- Overloading of functions and operators
- Classes with (multiple) inheritance
- Access control to class members (private, protected, public, friend)
- Constructors and Destructors for Classes
- Ambiguity resolution

- Static class member variables and static class member functions
- Constant class member variables and constant class member functions
- Declarations of variables are anywhere within a block of statements
- C++ comments
- References
- Inlining
- C linkage (extern "C")
- Standard predefined macro "__cplusplus"
- Explicit Constructor calls are creating temporary class objects
- Virtual functions and virtual base classes
- Pure virtual functions and abstract classes
- Templates
- Pointer to members
- Default Arguments
- Operators "new" and "delete"
- C++ libraries (streams, new/delete, strings, complex, bitset, STL)
- Special segments for virtual tables and compiler generated functions

The following C++ features are not supported:

- Static local class objects
- Non-constant initialization of global non-class objects
- Anonymous unions
- Goto-statement checks for initialization and Destructor calls
- Exception handling
- Namespaces
- Runtime Type Information (RTTI)
- Keywords bool, explicit, mutable, using, typeid
- Overloading with non-exact match
- Explicit operator calls
- Alternative C++ cast syntax
- Base class member access modification
- "namespace", "using", "bool", "mutable" keywords

- "const_cast", "static_cast", "reinterpret_cast"
- C++ libraries "exception" and "valarray"

Additional Keywords in C++

The following keywords are available in C++ programs:

```
class      operator     this
delete    private      throw
friend    protected   try
inline    public       catch
new       template    virtual
```

C Linkage

You can mix C and C++ sources using the guidelines defined in this section.

To call a C function from a C++ program, the function prototype must be declared with `extern "C"`. This lets the C++ Compiler know that the function is encoded in the "normal" C name encoding format.

Example:

```
extern "C" void f(void);

void main(void) {
    f(); /* call the C function f() */
}
```

Linkage specifications are used for linking functions written in different languages. The linkage specifications currently supported by this compiler are "C" and "C++".

Example:

```
extern "C" {
    void g(); // "C" linkage
extern "C++" {
    int f(); // "C++" linkage
}
}
```

The Standard Predefined Macro `__cplusplus`

The macro, ‘`__cplusplus`’, is automatically defined if the Compiler translates a C++ source. This macro distinguishes, on source level, between ANSI-C and C++ Compilers, and makes sources compatible for both for ANSI-C and C++ Compilers.

Example:

The source code inside this Compiler directive is compiled only if the source is compiled as a C++ program. Otherwise, it is ignored.

```
struct A {  
    int j;  
    #ifdef __cplusplus  
        void f(int i); /* only legal for C++ */  
    #endif  
} a;
```

Example:

The following example shows the use of the macro in ANSI-C header files to make them compatible for both ANSI-C and C++.

```
#ifdef __cplusplus  
    extern "C" { /* C++: use C calling convention */  
#endif  
  
void foo(int, double);  
/* ... */  
  
#ifdef __cplusplus  
    }  
#endif
```

Special Segment Support for C++

Programming in C++ requires special attention to program segmentation. Normally, you have control over data and code allocation, using pragmas, as in normal C programs. However, there are special segments used for C++.

VIRTUAL_TABLE_SEGMENT

The compiler generates virtual function tables if virtual functions are used. Because classes often are declared in header files, each implementation file including such

header files with classes containing virtual member functions, may generate virtual function tables. These tables are constant by default and may be allocated in ROM.

To simplify this, the compiler places all virtual tables into a special segment named VIRTUAL_TABLE_SEGMENT. You can use this in the linker parameter file to allocate the virtual tables into ROM:

```
DEFAULT_ROM, ROM_VAR, VIRTUAL_TABLE_SEGMENT INTO MY_ROM
```

Additionally, the linker uses this segment name to avoid duplicate definitions of virtual function tables in your linked application.

GEN_FUNCS_SEGMENT

The compiler generates some compiler-generated functions: copy constructors, default constructors and destructors.

All compiler-generated functions are placed into a special segment named GEN_FUNCS_SEGMENT. This segment may be used in the linker parameter file:

```
DEFAULT_ROM, GEN_FUNCS_SEGMENT INTO MY_ROM
```

Additionally, the linker uses this segment name to avoid duplicate definitions of compiler generated functions in your linked application.

Differences between ANSI-C and C++

Empty Parameter Declarations

A function such as f() has no parameters in C++. Although in C, it can contain any number of functions:

```
void foo();  
/* C++ : same as 'void foo(void)' */  
/* ANSI-C: undefined number of parameters */
```

Return Values

Functions returning a value must have a correct C++ return statement.

Example:

```
int f(void) {  
    int i;  
    i=3;  
} /* C legal, C++ error: return required. */
```

Implicit Parameter Declaration

Declare all functions before using them in C++.

Example:

```
void f(void) {  
    g(); /* ANSI-C legal, C++ error: missing prototype */  
}
```

See also Option [-Wpd](#)

Constant Variables

Initialize all constant variables in C++.

```
#ifdef __cplusplus  
    const int i = 3; /* C++ requires initialisation */  
#else  
    const int i; /* legal for ANSI-C */  
#endif
```

Additionally, constant objects without ‘extern’, have static linkage. Objects with static linkage cannot be used in other modules. You must declare C++ constant objects with ‘extern’ to implement external linkage for these objects. See also [Ellis, page 108] for details.

```
#ifdef __cplusplus  
    const int i = 3; // definition and static linkage  
    extern const int j; // declaration  
    const int j = 4; // definition and external linkage  
    static const int k = 3; // definition and static linkage  
#else  
    const int i = 3; /* definition and external linkage */  
    extern const int j; /* declaration */  
    const int j = 4; /* definition and external linkage */  
    static const int k = 3; // definition and static linkage  
#endif
```

Name Spaces

The name space of structure tags and typedefs differ between C and C++.

Empty Classes

The size of an empty struct/class is always one. The Linker cannot allocate an object with size zero. Non-empty classes derived from empty classes do not contain a dummy byte.

Example:

```
class EmptyClass {} ; // sizeof(EmptyClass) returns 1!
class DerivedFromEmpty : EmptyClass {
    int a;
};

int i, j;

void main(void) {
    i = sizeof(EmptyClass);          /* == 1 */
    j = sizeof(DerivedFromEmpty);   /* == sizeof(int) */
}
```

More Details about C++

For more details please refer to "Ellis/Stroustrup: The Annotated C++ Reference Manual".

C++ and Embedded Systems

C++ Compilers are known to produce inefficient code. However, it is possible to achieve small more efficient code. This chapter provides you with tips on how to write efficient C++ programs suitable to embedded systems development.

Depending on your need for efficient code, you can choose between the languages Full ANSI Draft C++, Embedded C++ (EC++), and compactC++ (cC++) using the [-C++](#) option.

- Full ANSI Draft C++ supports the whole C++ language.
- Embedded C++ (EC++) supports a constant subset of the C++ language. EC++ does not support templates, multiple inheritance, virtual base classes and exception handling.
- compactC++ (cC++) supports a configurable subset of the C++ language. You can configure this subset with the [-Cn](#) option. This enables you to select which features to switch off or on, depending on your needs.

About Compiler Generated Functions

The Compiler generates special member functions for classes where necessary. This increases the code size of a module. However, those generated functions are often not called in the application. The Linker discards functions that are not called in the application.

Alternately, you can avoid the Compiler generating those functions by using the [-Cn=Ctr](#) option. Use the compiler [-Cn=Ctr](#) option to not create compiler-defined functions in C++. You may also do this manually:

```
class A {}  
  
A::A(){} // default constructor  
A::A(const A&){} // copy constructor  
A& A::operator(const A&){} // assignment operator  
A::~A(){} // destructor
```

Use inlining or class templates (described in the following two sections) if the 'Default Copy Constructor' does not generate automatically:

Use Inlining

Declare all member functions with the inline specifier or write the function body inside the class declaration.

Example:

```
struct A {  
    inline void f(void) /* inlined function */  
    void g(void) { /* ... */ } /* inlined function */  
};  
  
void A::f(void) {  
}
```

NOTE Inline functions do have implicit static linkage.

Another way is to use the option [-Oi](#).

Use Class Templates

You can parameterize classes with templates. This enables you to avoid class Constructors that perform unnecessary tasks at runtime. See [Listing 1.20](#).

Listing 1.20 Using Templates

```
*****  
/* bad: */  
*****  
class A {  
    int *array;  
    A(int i) { // needs a lot of run time!!!  
        array = malloc(i);  
    }  
};  
  
A a1(4); // class A with array of 4 elements  
A a2(7); // class A with array of 7 elements  
  
*****  
/* good: */  
*****  
template<int i> class A {  
    int array[i];  
    A();  
};  
  
A<4> a1; // class A with array of 4 elements  
A<7> a2; // class A with array of 7 elements
```

Avoid Function Templates

Function templates produce a good deal of code. Each instantiation of a template function generates code for the whole function, along with specific template arguments.

Example:

```
template<class T> void f(T t) { /* ... */ }  
  
void g(void) {  
    char *chp;  
    f(4); // generate void f(int) { /* ... */ }  
    f(chp); // generate void f(char *) { /* ... */ }  
}
```

Reduce the Type for Virtual Function Table Delta

Virtual function tables are generated using virtual member functions. These tables contain a function pointer and an associated delta value that accesses the base class. The delta values have a (signed) 16-bit type, allowing class objects with a size of 32 kBytes each. You can change the delta value type, using the `-TvtD` option, to save ROM usage for function tables. For example, if all class objects are smaller or equal 127 bytes, change the virtual table delta type to a signed 1 byte type using [-TvtD](#). If there is a class object not fitting into this type, an error message ([C1393](#)) is generated.

Avoid Pointer to Member Parameters and Pointer to Member Returns

The compiler handles a data structure as the pointer to member type is not a pointer (see “The annotated C++ reference manual” from ELLIS-STROUSTRUP). Refer to [Listing 1.21](#).

Listing 1.21 Avoid Pointer Member Parameters and Pointer Member Returns

```
class A{
public:
    void fct(void){ /*.....*/ }
};

void f(void (A::*pm)(void)){
    A a;
    (a.*pm)();
}

void main(void){
    f(&A::fct);
}

// Produces the following behavior:
typedef ptrmbrType{
    int offset;
    int index;
    void *fctMbr;
};

class A{
public:
    void fct(void){ /*.....*/ }
};
```

```

void f(ptrmbrType pm){
    A a;
    if (!(pm.offset==0 && pm.index==0 && pm.fctMbr==NULL)){
        if (pm.index== -1){
            (a.*pm)(); // call function in a normal way
        } else {
            (a.*pm)(); // call function in a virtual way
        }
    }
}
void main(void){
    ptrmbrType dummy;
    dummy.offset = 0;
    dummy.index = -1;
    dummy.fctMbr = &A::fct;
    f(dummy); // call f function in a normal way
}

```

The same kind of behavior is generated when returning pointer to member. No information is available at the function call statement in the event of a pointer-to-member as a parameter, or as a return. Handling these tests at runtime addresses this issue.

Avoid Class Parameters and Class Returns

Every class parameter passing and class return requires a call to the copy constructor of the class if the class needs to call a constructor. These types of copy constructor calls are not efficient. It is more efficient to use pass references. See [Listing 1.22](#).

Listing 1.22 Avoid Class Parameters and Class Returns

```

*****
/* bad:                                     */
*****  

class A {
    A(void);
    A(A &);
};

A f(A a) {
    A ret;
    return ret; // call copy constructor for return b
}

```

```
void g(void) {
    A a, b;
    b=f(a); // call copy constructor for argument a
}

/* **** */
/* good: */
/* **** */

class A {
    A(void);
    A(A &);
};

void f(A &a, A &b) {
    A ret;
    b=ret;
}

void g(void) {
    A a, b;
    f(a, b); // pass the address of a and b
}
```

Avoid Virtual Functions

Virtual functions are the key feature of object oriented programming. Though they are very inefficient in use. It is recommended to avoid them.

Avoid Virtual Base Classes

Virtual base classes are not efficient. It is recommended to avoid them. See also [‘Short C++ tutorial’](#)

Avoid NULL-Check of Class Pointers

Operations with pointers to related classes always require separate NULL-checks before adding offsets from base classes to inherited classes. Avoid operations of pointers to different, but related classes.

Example:

```
class A {
    int i;
};

class AA {
    int j;
};

class B : A {
};

void f(void) {
    A *ap; // pointer to base class
    B *bp; // pointer to inheriting class
    ap=bp; // generates following code:
            // ap=(bp==0)?0:(A *)((char *)bp+delta(A));
}
```

Declare Near First Use

An object is initialized (constructed) the moment it is declared. If you don't have enough information to initialize an object until you are half way down the function, create it half way down the function where the system can correctly initialize it. Don't initialize it to an "empty" value at the top then "assign" it later as this affects runtime performance.

Examples:

```
//bad:
void f() {
    int i=0;
    /* ... */
    i=5;
    /* using i */
}

//good:
void f() {
    /* ... */
    int i=5;
    /* using i */
}
```

About Code Size

Object files are quite large, especially if the source code is trivial. A simple "hello world" program generates an executable that is larger than most people expect. The

reason is: the <iostream.h> library is quite large and consists of numerous classes and virtual functions. Using any part of it might pull in nearly all of the <iostream.h> code as a result of the interdependencies. However, the linker discards functions and objects which are not used. The size of the executable decreases in comparison to the object file.

C++ Overhead

[Table 1.44](#) gives an overview about the possible overhead (ROM/RAM) using a C++ feature compared with standard ANSI-C.

Table 1.44 Overhead of C++ Features

C++ Feature	Overhead
C++ comments (e.g. '/* */')	None
Default arguments (e.g. 'foo(int, int b=3)')	None, except that for calling 'foo' with 'foo(5)' will result in a call with an additional parameter 'foo(5,3)'.
classes	None, if no inheritance or virtual functions are used. An empty struct/class occupies one byte of memory (RAM) (not possible in ANSI-C), but a derived class from an empty class does not contain this dummy byte.
access control (public, private, protected)	None
virtual functions (virtual keyword)	Large as additional pointers in the class object (virtual base pointers) are used (RAM) and the virtual function tables has to be allocated (ROM). Additionally, most of the virtual function calls have to reference this virtual function tables (dereferenciation) which results in additional code (ROM) compared with normal function calls.
stronger type checking (name encoding)	None
templates	Large only for function templates, because each instantiation generates additional code (ROM).
C linkage (e.g. 'extern "C"')	None

Short C++ Tutorial

1. [Classes](#)
2. [Member Access Control](#)

3. [Special Member Functions](#)
4. [Virtual Functions](#)
5. [Virtual Base Classes](#)
6. [Templates](#)
7. [Exception Handling](#)
8. [Pure Virtual Functions / Abstract Classes](#)
9. [Overloading](#)
10. [Pointer to Members](#)
11. [Temporary Objects](#)

1. Classes

The C++ class concept is a generalization of the C notion of a structure. Said another way, the C concept of a structure is a simple variant of the C++ class concept. The C structures do not support member functions of any kind.

The difference between using the keywords "class" and "struct" lies only in the default access control of the members. The structure members and base classes are public by default. Class members are private by default.

Example:

```
struct A {  
    int i;      // i is a public member of A  
protected:  
    void f(); // is a protected member function of A  
};  
  
class B {  
    int i;      // is a private member of B  
protected:  
    void f(); // is a protected member function of B  
};
```

A class is derived from another class which is then called a base class of the derived class. The derived class inherits the properties of its base classes, including its data members and member functions. In addition, the derived class overrides virtual functions of its bases and declare additional data members, functions, and so on. Access to class members is checked for ambiguity.

Sharing about the base classes that make up a class is expressed using virtual base classes.

Classes are declared abstract to ensure they are used only as base classes.

Example:

```
struct A {
    int i;
}; // A contains member i.

struct B : A {
    int j;
}; // B contains members i (inherited from A) and j.

void main(void) {
    A a;
    B b;
    a.i=5;
    b.i=10;
    b.j=15;
}
```

Static Members

A data or function member of a class is declared “static” in the class declaration. There is only one copy of a static data member. This copy is shared by all objects of the class in a program.

Example:

```
class A {
public:
    static int s;
    int i;
};

int A::s; // instantiation of the static member
void f(void) {
    A a;
    a.i=5; // ok
    a.s=10; // ok
    A::i=15; // error
    A::s=20; // ok
}
```

Member Functions

In a non-static member function, the keyword “this” is a pointer to the object for which the function is called.

Example:

```
class A {  
    int i;  
    void f(int j) {  
        this->i=j;  
    }  
};
```

Implicit Casting of Classes

Class objects are assigned to base class objects. The base class part of the derived class object is copied to the base class object.

Pointers to class objects are assigned to pointers that point to base class objects. The pointer points to the base class part of the derived class.

Example:

```
class A {  
    int i;  
};  
  
class B : public A {  
    int j;  
};  
  
void main(void) {  
    A a;  
    A *ap=&a;  
    B b;  
    B *bp=&b;  
    a=b;      // same as a.j=b.j  
    ap=bp;    // same as ap=(A *)bp;  
    b=a;      // error  
    bp=ap;    // error  
}
```

2. Member Access Control

Private

A member of a class is private; its name used only by member functions, member initializers, and friends of the class in which it is declared.

Protected

Its name is used only by member functions, member initializers, and friends of the class in which it is declared; and by member functions and friends of classes derived from this class.

Public

Its name is used by any function or initializer.

The code in [Listing 1.23](#) demonstrates how the access control specifiers affect access to class methods and data members.

Listing 1.23 Example Code—Class Member Access Control

```
class A {  
    private:  
        int i;  
    protected:  
        int j;  
    public:  
        int k;  
        void f(void);  
};  
  
class B : A {  
    void f(void);  
};  
  
void A::f(void) {  
    this->i=5; // ok, private member accessed by  
               // member function of same class  
    this->j=5; // ok, protected member accessed by  
               // member function of same class  
    this->k=5; // ok, public member  
}  
  
void B::f(void) {  
    this->i=10; // error, private member accessed by  
               // member function of different class  
    this->j=10; // ok, protected member accessed by  
               // member function of class derived  
    this->k=10; // ok, public member  
}
```

```
void main(void) {
    A a;
    a.i=15; // error, private member accessed outside
    a.j=15; // error, protected member accessed outside
    a.k=15; // ok, public member
}
```

Private Base Class

Access specifiers can also be applied to base classes. If the base class is private and the public and protected members of the base class are private in the derived class, the private members cannot be accessed in the derived class.

Protected Base Class

The public and protected members of the base class are protected in the derived class. The private members cannot be accessed in the derived class.

Public Base Class

The public members of the base class are public in the derived class. The protected members of the base class are protected in the derived class. The private members cannot be accessed in the derived class.

The code in [Listing 1.24](#) illustrates the effect of public, protected, and private class inheritance.

Listing 1.24 Example Code—Public, Private, and Protected Inheritance

```
class A {
public:
    int i;
};

class B {
public:
    int j;
};

class C : public A, private B {
    int k;
};
```

```
void main(void) {
    C c;
    c.i=5;    // ok
    c.j=10;   // error
}
```

Friends

A friend of a class is a function that is not a member of the class but permitted to use the private and protected member names from the class. [Listing 1.25](#) shows an example.

Listing 1.25 Example Code—Using friend Functions

```
class A {
    int i;
    friend void f(void); // f() is a friend of A
    friend class X;    // all member functions of X are friends of A
};

class X {
    void f(void) {
        A a;
        a.i=10; // ok
    }
};

void f(void) {
    A a;
    a.i=5; // ok
}
```

3. Special Member Functions

Constructors, destructors and conversions are special member functions.

Constructors

The constructor of a class is called whenever an object of that class is created. The following tasks are performed:

- Assign vptr (pointer to virtual base class)

- Call constructors of base classes and member class objects (in declaration order)
- Assign vptr (pointer to virtual function table)
- Execute function body of constructor

The code in [Listing 1.26](#) shows this process.

Notes:

- Constructors are overloaded (“default constructors” contain no parameters).
- Global class objects call their constructors when all global objects get initialized.

Listing 1.26 Example Code—Constructors

```
struct A {  
    A(void) { /* function body */ } // Default constructor  
    A(int i) { /* function body */ }  
    A(A &a) { /* function body */ } // Copy constructor  
};  
  
struct B : A {  
    B(void) : A(5) { /* function body */ } // Default constructor  
};  
  
void f(void) {  
    A a;          // calls A::A(void)  
    A a2(5);     // calls A::A(int i)  
    B b;          // calls B::B(void), that calls A::A(int i)  
}
```

Destructors

The class destructor is called before an object of that class is destroyed.

The following tasks are performed:

- Execute function body of destructor
- Call destructors of base classes and member class objects (in reverse declaration order)

The code in [Listing 1.27](#) shows the order in which an object’s destructors are called.

Notes:

- Destructors cannot be overloaded.

- Global class objects call their destructors at the end of the program (after the main routine finished)

Listing 1.27 Example Code—Destructors

```
struct A {  
    ~A(void) { /* function body */ }  
};  
struct B : A {  
    ~B(void) { /* function body */ }  
};  
  
void f(void) {  
    A a;  
    B b;  
} // a object calls A::~A(void)  
// b object calls B::~B(void), that calls A::~A(void)
```

Conversion by Constructor

A constructor accepting a single argument specifies a conversion from its argument type to the type of its class.

Example:

```
class A {  
public:  
    A(int);  
};  
  
void main(void) {  
    A a = 1; // a = A(1)  
}
```

Conversion Functions

Conversion operators specify conversions from the owner class type to the type specified.

Example:

```
class A {  
operator char*(); // defines a conversion from A to char*  
};
```

Compiler Generated Functions

- Default constructor

The default constructor is generated if no constructor is declared by the programmer. It is also generated if there is a base or member class having a constructor or, there are virtual base classes or virtual member functions.

- Copy constructor

The copy constructor is generated if no copy constructor is declared by the programmer.

- Destructor

The default destructor is generated if no destructor is declared by the programmer, and if there is a base or member class having a destructor.

- Assignment operator

The assignment operator is generated if no assignment operator is declared by the programmer.

Sometimes special member functions are generated by the compiler in several compilation units (e.g if there is a class declaration in a header file included in several compilation units). A normal linker treats them as different static linkage functions. This is not efficient as the code of the same function exists more than once in the executable. The Smart Linker merges the same compiler generated function from several compilation units into one function.

4. Virtual Functions

A virtual function is a member function that is redefined in derived classes. You can call a virtual function for that object and execute the derived class's version of the function when you refer to a derived class object using a pointer or a reference to the base class. See [Listing 1.28](#).

Listing 1.28 Example Code—Calling Different Versions of a Virtual Function

```
class A{
public:
    virtual void f(void);
};

class B : public A{
public:
```

```
    void f(void);
};

A a;
B b;
A pta;

void main(void) {
    pta = &a;
    pta->f(); // calls A::f
    pta = &b;
    pta->f(); // calls B::f
}
```

The **virtual** keyword is needed only in the base class's declaration of the function; any subsequent declarations in derived classes are virtual by default.

A derived class's version of a virtual function must have the same parameter list and return type as those of the base class. If these are different, the function is not considered a redefinition of the virtual function.

“Virtual functions” are functions that ensure the correct function is called for an object, regardless of the expression used to make the function call.

Suppose a base class contains a function declared virtual and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class.

Functions in derived classes override virtual functions in base classes only if their type is the same. A function in a derived class cannot differ from a virtual function in a base class in its return type only; the argument list must differ as well.

When calling a function using pointers or references, the following rules apply:

- A call to a virtual function is resolved according to the underlying type of object for which it is called.
- A call to a non-virtual function is resolved according to the type of the pointer or reference.

Because virtual functions are called only for objects of class types, you cannot declare global or static functions as **virtual**.

The **virtual** keyword is used when declaring overriding functions in a derived class, but it is unnecessary; overrides of virtual functions are always virtual.

Virtual functions in a base class must be defined unless they are declared using the *pure-specifier*.

The virtual function call mechanism is suppressed by explicitly qualifying the function name using the scope-resolution operator ‘::’. See [Listing 1.29](#).

Listing 1.29 Example Code—Suppressing the Virtual Function Call Mechanism

```
class A{
public:
    A(void);                  //constructor
    virtual void f(void);    //virtual function member
    virtual void g(void);    //virtual function member
};
void A::A(void){...} /*implementation of constructor for A.*/
void A::f(void){...} //implementation of f for A.
void A::g(void){...} //implementation of g for A.
class B : public A{
private:
    void g(void);
};
void B::g(void){...} //implementation of g for B.

void main(void) {
    B *ptb = new B();
    ptb->A::g(); /*explicit qualification call A::g.*/
    A *pta = ptb;
    pta->A::g(); /*explicit qualification call A::g.*/
}
```

Both calls to `g()` in the preceding example suppress the virtual function-call mechanism.

Access to Virtual Functions

The access control applied to virtual functions is determined by the type used to make the function call. Overriding declarations of the function does not affect the access control for a given type. See [Listing 1.30](#).

Listing 1.30 Example Code—Access Control and Virtual Functions

```
class A{
public:
    virtual void f(void);
};

class B : public A{
```

```
private:  
    void f(void);  
};  
  
void main(void) {  
    B b;  
    A *pta = &b;  
    B *ptb = &b;  
    pta->f(); // f is public.  
    ptb->f(); // f is private, ERROR  
}
```

NOTE The virtual function f is called using a pointer to the base class A. This does not mean that the function called is the base-class version of that function.

Function Specifiers

Virtual Specifier

The **virtual** keyword is applied only to non-static class member functions. It signifies that binding of calls to the function is deferred until run time.

- Advantage

Derived classes and virtual functions are the key to the design of many C++ programs.

The use of derived classes and virtual functions is often called object-oriented programming. Furthermore, the ability to call a variety of functions using exactly the same interface, as is provided by virtual functions, is sometimes called polymorphism.

- Disadvantage

Such characteristics require more memory (many additional objects), more compilation time (many additional objects to manage) and generate more code (many additional objects to handle).

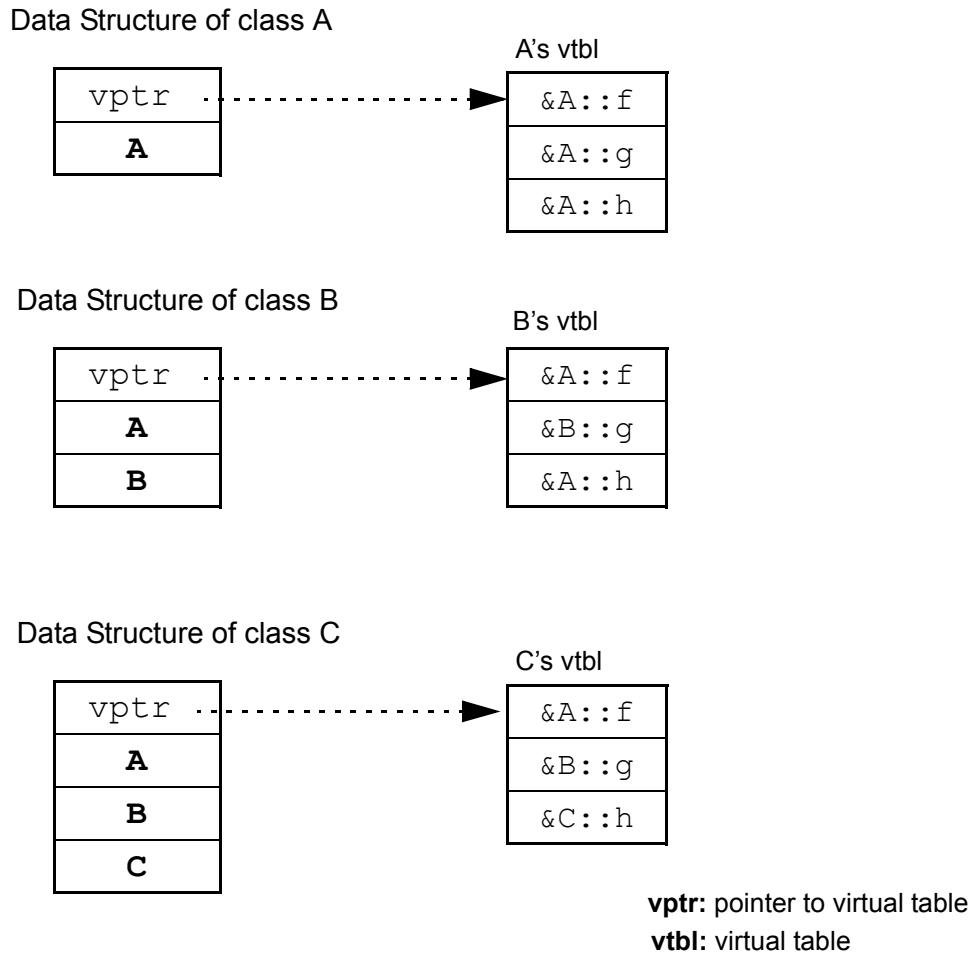
- Implementation

The implementation of the virtual-functions mechanism is based on the following concept. Virtual functions are implemented with a table of pointers to virtual functions, the vtbl. A pointer object is introduced in the class scope, the vptr. It points to the vtbl.

Single Inheritance and Virtual Functions

[Figure 1.2](#) shows the virtual tables and other data structures produced by the code shown in [Listing 1.31](#).

Figure 1.1 Single Inheritance—vtbl Layout Produced by Example Code



Listing 1.31 Example Code—Single Inheritance and Virtual Functions

```
class A{
public:
    virtual void f(int);
    virtual void g(int);
    virtual void h(int);
};
```

```
class B : public A{
public:
    void g(int);
};

class C : public B{
public:
    void h(int);
};
```

A call to a virtual function is transformed by the Compiler into an indirect call. For example:

```
C *pc = new C;
pc->g(2);
```

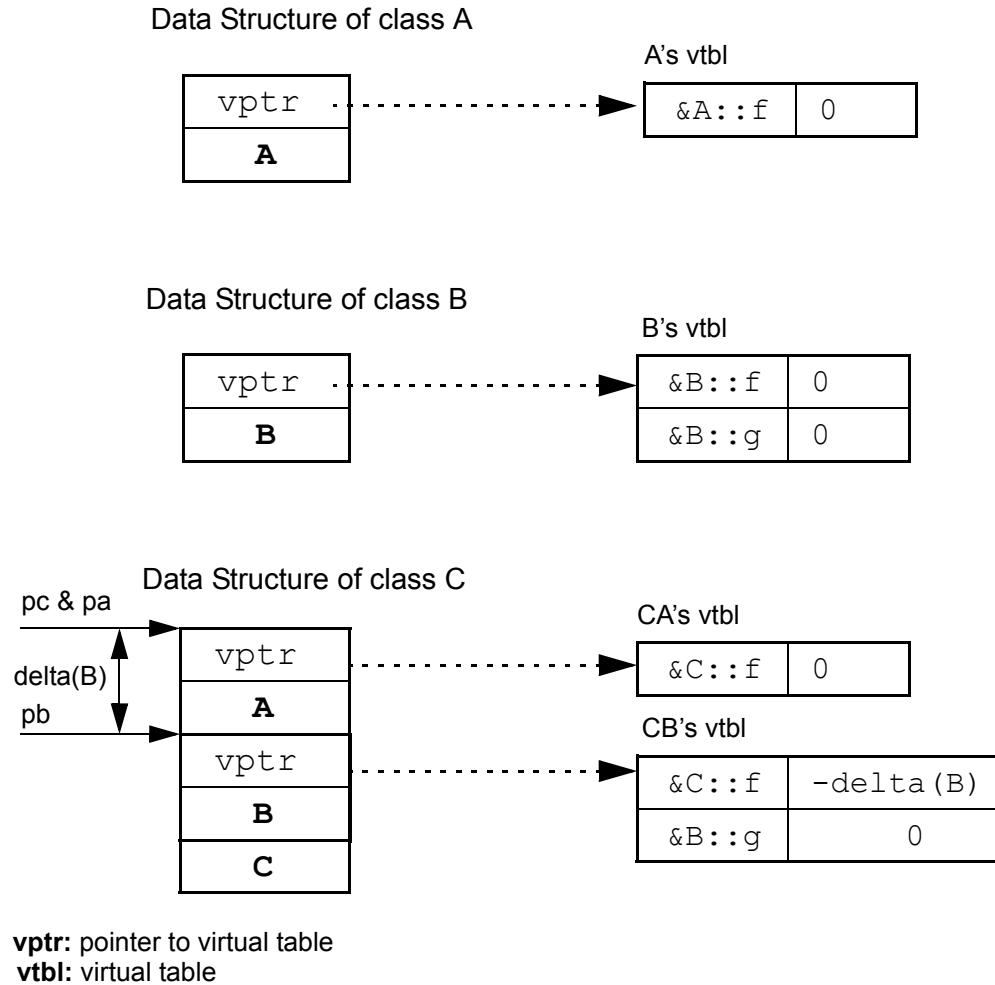
becomes something like:

```
(* (pc->vptr[1]))(pc, 2);
```

Multiple Inheritance and Virtual Functions

[Figure 1.2](#) shows the virtual tables and other data structures produced by the code shown in [Listing 1.32](#).

Figure 1.2 Multiple Inheritance—vtbl Layout Produced by Example Code



Listing 1.32 Example Code—Multiple Inheritance and Virtual Functions

```
class A{
public:
    virtual void f(int);
};

class B{
public:
    virtual void f(int);
    virtual void g(int);
};
```

```
class C : public A, public B{
public:
    void f(int);
};

C *pc = new C;
A *pa = pc;
B *pb = pc;
```

Because class C derived from class A and from class B, the following calls will all invoke C::f():

```
pa->f();
pb->f();
pc->f();
```

On entry to C::f(), the this pointer must point to the beginning of the C object, and not to the B part. It cannot always be known at compile time, however, that the B pointed by pb is part of the C, so the offset of a B object within a C object, delta(B), is not a constant at compile time. Consequently delta(B) must be stored where it is found at run time. Because the offset is used only for calling a virtual function, the logical place to store it is in the virtual function table.

The call

```
pb->f();
```

in the example above invokes C::f(), having been transformed by the Compiler to something like this:

```
register vtbl_entry *vt =
&pb->vtbl[index(f)];
(*vt->fct)((B *)((char *)pb+vt->delta));
```

Virtual Tables Naming Convention

In some C++ implementation, the code for a virtual function table is duplicated in the object file produced for each C++ source file that declares or includes declarations of the class containing the virtual functions.

Now suppose that these declarations are included in multiple C++ source files. Each time a Compiler processes these declarations, even if it knows the declarations came from a header file, it will not know whether the virtual function tables have already been generated in a separate compilation. Therefore, it must generate the tables.

Because most Linkers will report an error if an identifier is defined more than once, virtual function tables must have unique names. Unique names are created by including the name of the source file in the name of the table. With the declarations

above, when included in the files windows.c and interface.c, might yield tables named as follows:

```
____vtbl__A__windows_c
____vtbl__B__windows_c
____vtbl__CA__windows_c
____vtbl__CB__windows_c
____vtbl__A__interface_c
____vtbl__B__interface_c
____vtbl__CA__interface_c
____vtbl__CB__interface_c
```

The Linker supports merging virtual tables with the same name in different files. So the filename does not have to be encoded into the virtual table name. This results in having only one virtual table in memory for each class, even if the class is used in different compilation units.

5. Virtual Base Classes

Multiple Base Classes

A class is derived from any number of base classes. The use of more than one direct base class is called multiple inheritance.

The order of derivation is not significant except as specified by the semantics of initialization by constructor, cleanup, and storage layout.

A class shall not be specified as a direct base class of a derived class more than once. But you can assign it as an indirect base class more than once. See [Listing 1.33](#).

Listing 1.33 Example Code—Multiple Inheritance

```
class A{
public:
    // member list
};

class B : public A, public A{ //Ill-formed
public:
    // member list
};
```

```
class A{
public:
    int next;
    // member list
};

class B : public A {
public:
    // member list
};

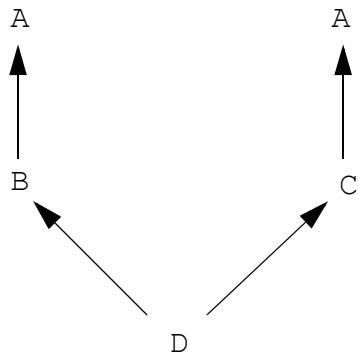
class C : public A {
public:
    // member list
};

class D : public B, public C { //Well-formed
public:
    void f(){}
    // member list
};
```

A base class specifier that does not contain the keyword **virtual**, specifies a *non-virtual* base class. A base class specifier that contains the keyword **virtual**, specifies a *virtual* base class. For each distinct occurrence of a *non-virtual* base class in the class lattice of the most derived class, the complete object shall contain a corresponding distinct base class sub-object of that type. For each distinct base class that is specified *virtual*, the complete object shall contain a single base class sub-object of that type.

Example:

For an object of class type C, each distinct occurrence of a (non-virtual) base class A in the class lattice of C corresponds one to one with a distinct A sub-object within the object of the type C. Given the class D defined above, an object of class D will have two sub-objects of class A as shown below.



In such lattices, explicit qualification is used to specify which sub-object is meant. The body of the function

```
C::f() { A::next = B::next; } // Well-formed
```

Without the A:: or B:: qualifiers, the definition of C::f above would be ill-formed because of ambiguity.

For another example:

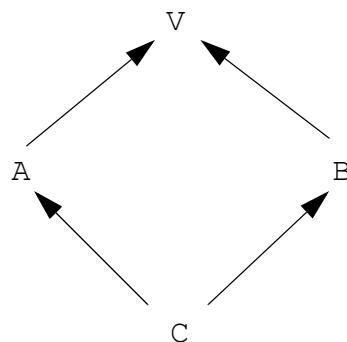
```
class V{
public:
    // member list
};

class A : public virtual V {
public:
    // member list
};

class B : public virtual V {
public:
    // member list
};

class C : public A, public B {
public:
    // member list
};
```

For an object *c* of class type C, a single sub-object of type V is shared by every base sub-object of *c* that is declared to have a *virtual* base class of type V. Given the class C defined above, an object of class C will have one sub-object of class V, as shown below.



For another example:

A class can have both virtual and non-virtual base classes of a given type.

```
class A{
public:
    // member list
};

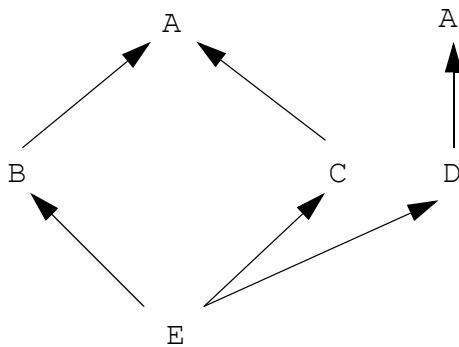
class B : public virtual A {
public:
    // member list
};

class C : public virtual A {
public:
    // member list
};

class D : public A {
public:
    // member list
};

class E : public B, public C, public D {
public:
    // member list
};
```

For an object of class type E, all *virtual* occurrences of base class A in the lattice of E correspond to a single A sub-object within the object of type E, and every other occurrence of a (non-virtual) base class A in the lattice of E correspond one to one with a distinct A sub-object within the object of type E. Given the class E defined above, class E has two sub-objects of class A:D's A and the virtual A shared by B and C, as shown below.



Implementation

Suppose we have the following inheritance hierarchy.

```

class V{
public:
    // member list
};

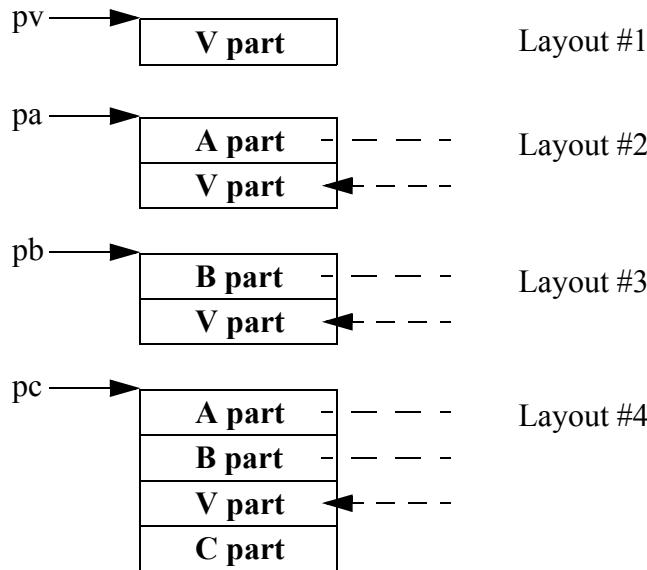
class A : public virtual V {
public:
    // member list
};

class B : public virtual V {
public:
    // member list
};

class C : public A, public B {
public:
    // member list
};

```

Each A object or B object will contain an V, but only one object of class V will exist in a C object. Clearly the object representing the virtual base class V cannot be in the same position relative to both A and B in all objects. Therefore a pointer to V must be stored in all objects of classes that have V as a virtual base. An implementation of A, B and C objects will look something like the figure below.



The virtualness of V in A, B and C is a property of the derivation, and not a property of the V itself.

To obtain such a memory layout (see above #4), the Compiler has to know that constructors of A and B classes have not to call V class constructor, whereas they have to call V class constructor to obtain layouts #2 and #3. To give the Compiler this information, an hidden global variable called “StandAlone” is used, it is defined as one byte type in memory. StandAlone is set to 1 if A or B classes are not base classes, then constructor of virtual base class V will be called. StandAlone is set to 0 if A or B classes are base classes (here of class C), then constructor of virtual base class V won’t be called. The Compiler is implemented this way:

```
oneByteType StandAlone
set StandAlone to 1
...
if current class contains base class(es) and not a VBPT
    for each base class
        if current base class contains a VBPT
            set StandAlone to 0
    ...
for each constructor
    if constructor of class containing a VBPT
        for each base class
            if virtual base class and StandAlone is set to 1
                call constructor
            else if not virtual base class
                call constructor
        else
            call constructor
```

Except that it results in a unique object in its derived classes, a virtual base class behaves the same way a non-virtual base class does. Every virtual base of a given class type in an inheritance structure. Given the declarations in [Listing 1.34](#) ...

Listing 1.34 Example Code—Virtual Base Classes

```
class V{
public:
    // member list
};

class A : public virtual V {
public:
    // member list
};

class B : public virtual V {
public:
```

```
// member list
};

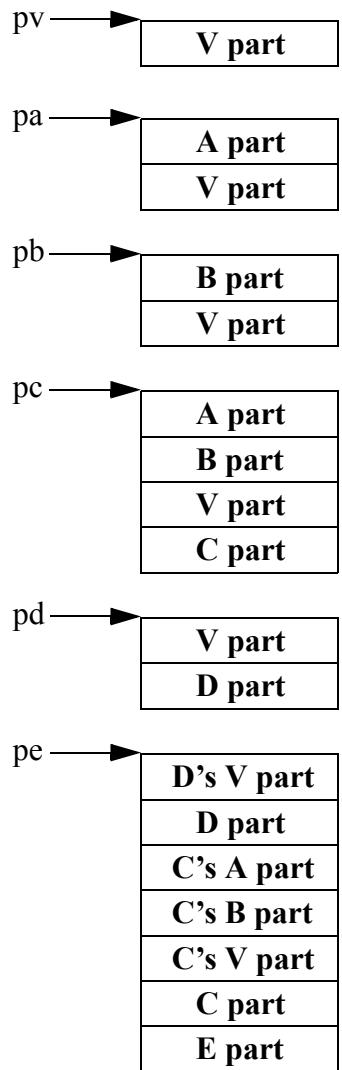
class C : public A, public B {
public:
    // member list
};

class D : public V {
public:
    // member list
};

class E : public D, public C {
public:
    // member list
};
```

an E object will contain two V objects, one virtual and one non-virtual. An implementation of an E object will look something like the illustration in [Figure 1.3](#).

Figure 1.3 Layout of an Object of Class E



Virtual Base Classes and Casting

One can cast from a derived class to a virtual base class. The generated code will use the pointer to the virtual base class stored in the derived class.

Casting from a virtual base class to a derived class is disallowed to avoid requiring an implementation to maintain pointers to enclosing objects.

Multiple Access

In multiple-inheritance lattices involving virtual base classes, a given name is reached through more than one path. Because different access control is applied along these different paths, the Compiler chooses the path that gives the most access.

```
class V{
public:
    // member list
};

class A : private virtual V {
public:
    // member list
};

class B : public virtual V {
public:
    // member list
};

class C : public A, public B {
public:
    // member list
};
```

In the figure above, a name declared in class V is always reached through class B. The right path is more accessible because B declares V as a public base class, whereas A declares V as private.

6. Templates

A template defines a family of types or functions.

Class Templates

A class template specifies how individual classes are constructed.

Example: Several lists with elements of different types should be constructed.

```
template <class T> class List {
    T      data;
    List *next;
};

List<int> li;      // generate list class
                  // where T stands for int
List<char *> lcp; // generate list class
                   // where T stands for pointer to char
```

Function Templates

A function template defines an unbounded set of related functions. You must use every template argument in the argument types of a function template.

Example: Several sorting functions for different types are constructed.

```
template <class T> void sort(T elem) {
    /* ... */
}

void main() {
    Type1 t1;
    Type2 t2;

    sort(t1); // generate sort function
               // where T stands for Type1
    sort(t2); // generate sort function
               // where T stands for Type2
}
```

Member Function Templates

A member function of a template class is implicitly a template function with the template arguments of its class as its template arguments.

Example:

```
template <class T> class A {
    T f(void);
};

template <class T> T A<T>::f(void) {
    /* ... */
}
```

Template Arguments

There are two kinds of template arguments: type specification and expression specification. See [Listing 1.35](#).

Listing 1.35 Templates—Type Specification and Expression Specification

```
template <class T> /* type specification */
class A {
    T member;
};
```

```
A<int> ai;    // generate A with a member of type int
A<int *> ai; // generate A with a member of type pointer to int

template <int i> /* expression specification */
class B {
    int array[i];
};

B<5> b5;      // generate B with an array of 5 elements
B<4*20> b80; // generate B with an array of 80 elements
```

Static Members and Variables

Each template class or function generated from a template has its own copies of any static variables or members. See [Listing 1.36](#).

Listing 1.36 Templates—Static Members and Variables

```
template <class T> class X {
    static T s;
};

X<int> aa;      // generate X with static member of type int
X<char *> bb;  // generate X with static member of type char*

template <class T> f(T p) {
    static T s;
    /* ... */
}

void g(int a, char *b) {
    f(a); // generate f() with local static of type int
    f(b); // generate f() with local static of type char*
}
```

Declarations and Definitions

There must be exactly one definition (provided by the programmer) for each template of a given name in a compilation unit (for classes and static linkage functions), or in a program (for external linkage functions). T

here are many declarations. The definition generates specific template classes and template functions to match the uses of the template. Using a template class name constitutes a declaration of template class.

Template classes must be defined before being used. Template functions do not have to be defined before using. If a specific function could not be generated from a template function definition, the linker will tell, which definition is needed. See [Listing 1.37](#).

Listing 1.37 Example Code—Link source1.cpp with source2.cpp

```
/* source1.cpp */
template <class A> void f(A i); // declaration, not definition

void main() {
    f(3); // Compiler does not generate f(), but no error !
}

/* source2.cpp */
template <class A> void f(A i) { // definition
    /* ... */
}

static void dummy() {
    /* dummy function for generating function from the template */
    f(3); // without this, the linker would complain !
}
```

7. Exception Handling

Exception handling is not implemented yet.

8. Pure Virtual Functions / Abstract Classes

Pure Virtual Functions

A virtual function is specified pure by using a pure-specifier (“=0”) in the function declaration and in the class declaration.

```
class A{
public:
    virtual void f(void) = 0; // pure virtual function.
};
```

A definition for a pure virtual function is not needed unless explicitly called with the qualified-id syntax (nested-name-specifier templateopt unqualified-id).

```
class A{
public:
    virtual void f(void) = 0;
};

class B : public A{
public:
    void f(void){ int local=0; }
};

void main(void){
    B b;
    b.A::f(); //causes link error because no object is defined.
    b.f();   //call the function defined in B class.
}
```

Pure virtual functions make up the class containing an abstract class. Abstract class mechanisms support the notion of a general concept, of which only more concrete variants can actually be used. An abstract class also defines an interface for which derived classes provide a variety of implementations.

Pure virtual functions are the base of the abstract class concept.

Abstract Classes

The abstract class mechanism defines a general concept, such as a shape, through which more concrete variants, such as circle and square, are defined. An abstract class can also define an interface for which derived classes provide a variety of implementations.

An abstract class is a class that is used only as a base class of some other class. No objects of an abstract class may be created except as objects representing a base class of a class derived from it. A class is abstract if it has at least one pure virtual function.

A virtual function is specified pure by using a pure-specifier (= 0) in the function declaration and in the class declaration. You must define a pure virtual function explicitly, calling it with the qualified-name syntax (ClassName::Member).

Example:

```
class A{
public:
    virtual void f(void) = 0;
    //...
};
```

```
class B : public A{
public:
    void f(void){}
};
```

An abstract class may not be used as an argument type, as a function return type, or as the type of an explicit conversion. Pointers and references to an abstract class may be declared.

Example:

```
A a;           // ERROR
A fct(void);   // ERROR
void fct(A);   // ERROR
B b;
A *pa = &(A)b; // ERROR
A *pa;         // OK
A *fct(void); // OK
void fct(A *); // OK
A *pa = (A *)b; // OK
```

Pure virtual functions are inherited as pure virtual functions.

Example:

```
class A{
public:
    virtual void f(void) = 0;
    virtual void g(void) = 0;
    ...
};

class B : public A{
public:
    void f(void){}
    // void B::g(void) is a pure virtual function
};
```

Since A::g() is a pure virtual function B::g() is a pure virtual by default. The alternative declaration is:

```
class B : public A{
public:
    void f(void){}
    void g(void); // must be define somewhere
};
```

This would make class B non-abstract. A definition of B::g() must be provided somewhere in your code.

Member functions are called from a constructor of an abstract class. An error is generated if a pure virtual function is called directly or indirectly for the object being created from such a constructor.

```
class A {  
public:  
    virtual void f(void) = 0;  
    A(){  
        f(); // ERROR  
    }  
};
```

You can define that as a pure virtual. The virtual is called using explicit qualification only, as follows:

```
class A {  
public:  
    virtual void f(void) = 0;  
    A(){  
        f(); // ERROR  
    }  
    void A::f(void){  
        //...  
    };
```

Abstract classes with virtual base classes.

See [Listing 1.38](#), [Listing 1.39](#), [Listing 1.40](#), and [Listing 1.41](#).

Listing 1.38 Example 1: Abstract Classes with Virtual Base Classes

```
class A {  
public:  
    virtual void f(void) = 0;  
    virtual void g(void) = 0;  
    virtual void h(void) = 0;  
    //...  
};  
  
class B : virtual public A{  
public:  
    void g(void){}  
};  
  
class C : virtual public A{
```

```
public:  
    void g(void){}  
};  
  
class D : public A, public B{  
public:  
    void h(void){}  
};
```

A class is abstract: contains 3 pure virtual functions (f, g, h)

B class is abstract: contains 2 pure virtual functions (f, h)

C class is abstract: contains 2 pure virtual functions (f, h)

D class is abstract: contains 1 pure virtual function (f)

Listing 1.39 Example 2: Abstract Classes with Virtual Base Classes

```
class A{  
public:  
    virtual void f(void) = 0;  
    virtual void g(void) = 0;  
    virtual void h(void) = 0;  
    //...  
};  
  
class B : virtual public A{  
public:  
    void f(void){}  
};  
  
class C : virtual public A{  
public:  
    void g(void){}  
};  
  
class D : public A, public B{  
public:  
    void h(void){}  
};
```

A class is abstract: contains 3 pure virtual functions (f, g, h)

B class is abstract: contains 2 pure virtual functions (g, h)

C class is abstract: contains 2 pure virtual functions (f, h)

D class is not abstract

Listing 1.40 Example 3: Abstract Classes with Virtual Base Classes

```
class A{
public:
    virtual void f(void) = 0;
    virtual void g(void) = 0;
    virtual void h(void) = 0;
    //...
};

class B : public A{
public:
    void g(void){}
};

class C : public A{
public:
    void g(void){}
};

class D : public A, public B{
public:
    void h(void){}
};
```

A class is abstract: contains 3 pure virtual functions (f, g, h)

B class is abstract: contains 2 pure virtual functions (f, h)

C class is abstract: contains 2 pure virtual functions (f, h)

D class is abstract: contains 2 pure virtual functions (B::f, C::f)

Listing 1.41 Example 4: Abstract Classes with Virtual Base Classes

```
class A{
public:
    virtual void f(void) = 0;
    virtual void g(void) = 0;
    virtual void h(void) = 0;
    //...
};
```

```
class B : public A{
    public:
        void f(void){}
};

class C : public A{
    public:
        void g(void){}
};

class D : public A, public B{
    public:
        void h(void){}
};
```

A class is abstract: contains 3 pure virtual functions (f, g, h)

B class is abstract: contains 2 pure virtual functions (g, h)

C class is abstract: contains 2 pure virtual functions (f, h)

D class is abstract: contains 2 pure virtual functions (B::g, C::f)

9. Overloading

Function Overloading

Overloading allows multiple functions with the same name to be defined provided their argument lists differ sufficiently for calls to be resolved.

Example:

```
void f(int i);
void f(void);
void g(void) {
    f(5);    // call f(int i)
    f();     // call f(void)
}
```

Operator Overloading

By overloading operators, you can redefine the meaning of most C++ operators when at least one operand is a class object.

Example:

```
class X {
    X operator + (int);
}
void g(X x) {
    X a;
    a=x+10; // call X::operator + (int)
}
```

Default Arguments

If an expression is specified in an argument declaration this expression is used as a default argument. All subsequent arguments must have default arguments supplied in this or previous declarations of this function.

Default arguments are used in calls where trailing arguments are missing. A default argument cannot be redefined by a later declaration (not even to the same value).

Example:

The declaration

```
point(int = 3, int = 4);
```

declares a function, that is called in any of these ways:

```
point(1,2);
point(1);      // equivalent to point(1,4);
point();       // equivalent to point(3,4);
```

Operator new and delete

The "new" operator attempts to create an object of the type specified.

If the type is a class having a constructor, the object is created only if suitable arguments are provided, or if the class has a default constructor. First, the "new" operator is called. Then the constructor is called if necessary. If the type is a class array, the default constructor of each element is called. The compiler generates a loop going through all array elements.

[Listing 1.42](#) shows code that uses the global new operator.

Listing 1.42 Using the Global new Operator

```
class A {
/* ... */
A();
A(int, int);
```

```
};

void f() {
    char *cp = new char; // create a "char" object
    int *ip = new int[3]; // create an "array of int" object
    A *ap = new A(4,3); // create a "A" object with Constructor
                         // call to A::A(int,int)
    A *aap = new A[3]; // create an "array of A" object with
                       // default Constructor call for each element
}
```

The "new" operator is defined globally or within a class. The global declaration is done internal to the compiler. The member operator "new" is a static member.

[Listing 1.43](#) shows code that defines two versions of new operator within a class and then uses each definition of this operator.

Listing 1.43 Implementing the new Operator Within a Class

```
void *operator new(size_t size) {

}

class A {
    void *operator new(size_t size) {
    }
    void *operator new(size_t size, int flag) {
    }
};

void f(void) {
    A *a = new(10) A; // calls A::operator new(size_t, int)
                      // with parameter "flag"=10
    A *a2 = new A;    // calls A::operator new(size_t)
}
```

The "delete" operator destroys an object created by the new operator.

If the type of the object is a class having a destructor, the destructor has to be called before the call of the "delete" operator. If the type is a class array having a destructor, the destructor of each element must be called. Therefore the compiler generates a loop going through all array elements. The "delete" operator is defined globally or within a class.

[Listing 1.44](#) shows code that uses the global delete operator.

Listing 1.44 Using the Global delete Operator

```
class A {
    /* ... */
    ~A();
}

void f(char *cp, int *ip, A *ap, A *aap) {
    delete cp;          // ok
    delete[] cp;        // ok
    delete[3] cp;       // ok, ignoring number of elements

    delete ip;          // ok
    delete[] ip;        // ok
    delete[3] ip;       // ok, ignoring number of elements

    delete ap;          // ok, calls once the destructor
    delete[] ap;        // error, don't know, how many destructor calls
    delete[3] ap;       // ok, calls 3 times the destructor
}
```

The "delete" operator is defined globally or within a class. The global declaration is already done internally in the compiler. The member operator "delete" is a static member.

[Listing 1.45](#) shows code that defines a delete operator within a class and then uses this definition.

Listing 1.45 Implementing the delete Operator Inside a Class

```
void operator delete(void *p) {
}

class A {
public:
    void operator delete(void *p) {
    }
};

class B {
public:
    void operator delete(void *p, size_t size) {
    }
};
```

```
void f() {
    B *b;
    delete b; // calls B::operator delete(void *, size_t)
               // with "size"=sizeof(B)
}
```

The global "new" and "delete" operators are implemented in the C++ library. A program calling new and delete needs to be linked with the appropriate library. Otherwise, those operators must be implemented by the programmer.

It does not matter to the Compiler if you do not provide any implementation of "new" and "delete" to your program as the global operators are internally declared (internal to the compiler). But the linker may not operate correctly if it does not find any implementation of "new" and "delete".

10. Pointer to Members

The C++ language provides no way of expressing the concept of a pointer to a member of a class. When a pointer to a member function is needed in an error handling function (for example), you had to subvert the language's type checking.

The type of a nonmember function:

```
int fct(char *),
```

An example is:

```
int (char *),
```

A pointer to such a function is of type:

```
int (*) (char *).
```

Pointers to nonmember functions are declared and used as:

```
void fct(char *);      //declare function
fct("Hello");         //call

void (*pfct)(char *); //declare pointer to function
(pfct) ("Goodbye");  //call through pointer
```

Consider a trivial class for which the member functions, fctMbr and object mbr, are and an object (obj) of that class are declared as follows:

```
class A{
    int fctMbr(char *);
    int mbr;
};
```

```

int A::fctMbr(char *) {
    /*...*/
}
A obj;

```

One can take the address of the member mbr in the object obj, as follows:

```
int *pMbr = &obj.mbr; // pMbr points to A's member mbr
```

The notation for the pointer to member of class A is A::* . The notation for taking the address of a member of a class A is &A:: . You can take the address of a member mbr in class A as follows:

```

int A::* spMbr = &A::mbr; // spMbr gets offset
                          // of 'mbr' in an A

```

Note that &obj.mbr yields the address of mbr in the specific object obj. Whereas &A::mbr yields a representation of mbr's relative position in all objects of class A.

It is not legal to take the address of a member that does not have a name. For example:

```

class X {
public:
    int a[2];
};

int X::*ai = &X::a[1]; // error: can't take address
                      // of unnamed member.

```

In a definition, the member function fctMbr of class A declared above appears as:

```

int A::fctMbr(char *) {
    /*...*/
}

```

The type of A::fctMbr is now expressed as int A::(char *). That is, “member of A that is a function taking a char * argument and returning an int”. A pointer to such a function is of type int (A::*)(char *). You can write this as:

```

// declare and initialize pointer to member function
int (A::*pmf)(char *) = &A::f;

// create an instance of A
A obj;

// call function through pointer for the object obj
int j = (obj.*pmf)("hello");

```

The above syntax is consistent with the C declarator syntax.

A pointer to member function is called through a pointer to an object as follows:

```
A *pObj;  
// call function through pointer for the object *pObj  
int k = (pObj->*pmf) ("Goodbye!");
```

A virtual function is called through a pointer to an object. See [Listing 1.46](#).

Listing 1.46 Calling a Virtual Function Through a Pointer to an Object

```
class B{  
    virtual vf();  
};  
  
class C : B{  
    vf();  
};  
  
int f2(B *pb, int (B::*pbf)()) {  
    //call virtual function pointed to  
    //by pbf for object that pb points to  
    return (pb->*pbf)();  
}  
  
void f1(){  
    C c;  
    //call f2 and pass pointer to derived object  
    //and pointer to virtual function in base class  
    int i = f2(&c, &B::vf);  
}
```

In f2, as called by f1 shown here, C::vf() is invoked. The implementation looks in the C table of virtual functions exactly as it would search for a call to a virtual function identified by name rather than by pointer.

Pointers to Static Members

Pointers to static members are ordinary pointers. The pointer to member syntax is not used with the static members. The address of a static member is taken with the pointer syntax for non-member objects. Similarly, a pointer that points to a static member is dereferenced as an ordinary pointer.

11. Temporary objects

In some circumstances, it may be necessary or convenient for the compiler to generate a temporary object. Such introduction of temporaries is implementation-dependent. When a compiler introduces a temporary object of a class that has a constructor, it must ensure that a constructor is called for the temporary object. Similarly, the destructor must be called for a temporary object of a class where a destructor is declared.

- Assigning constants to references:

```
void f(void) {
    int const & ri = 42;
    // ri points to a temporary holding the value 42
}
```

- Explicit constructor calls:

```
class A {
public:
    A();
    ~A();
};

void f(void) {
    A(); // a temporary is created.
    // Its address is passed to A::A()
} // dtor of the temporary is called here at the latest
```

- Functions returning a class type:

```
class A {
    A();
    ~A();
};

A f(void);

void g(void) {
    A a;
    a=f(); // return value of f() is copied to a temporary
    // with the copy constructor,
    // then the temporary is assigned to "a"
} // at the latest here the destructors of "a"
// and of the created temporary are called

void h(void) {
    A &a=f(); // return value of f() copied to a temporary
    // with the copy constructor,
```

```
// then the address of the temporary is assigned to the
reference
} // at the latest here the destructor
// of the created temporary is called
```

C++ Name Encoding, Type-safe Linkage

One of the advantages of using C++ is having safer type linkage than in ANSI-C. In ANSI-C, linkage is done by name. The Linker has no chance to detect a type mismatch (see also [“Implicit Parameter Declarations”](#)).

In C++ the parameter types (NOT the return type) of each function are encoded into the function name. The C++ Compiler does not encode function names if the name of the function is ‘main’, or if the function is defined or declared with a C linkage (extern “C”):

```
int foo2(void) {
    /* name in object file is 'foo2__F'
}
extern "C" int foo(int a, double d) {
    /* name in object file is 'foo'
}
void main(void) {
    /* name in object file is 'main' */
}
```

The C++ object files have different object name encoding than the ANSI-C object files. If you link C++ object files with ANSI-C object files, the imported functions must contain the linkage of the language from where they are exported. See the following example:

```
/* a.c (ANSI-C) */
void f(void) {
    /* ... */
}

/* b.cpp (C++) */
extern "C" void f(void);
    /* f() has "C" linkage, because exported from a ANSI-C
source */

void main(void) {
    f();
}
```

Type	Encoding
void	v
char	c
wide char	w
short	s
int	i
long	l
long long int	x
float	f
double	d
long double	r
long long double	r
generic 1 byte type	b1
generic 2 byte type	b2
generic 3 byte type	b3
generic 4 byte type	b4
generic 5 byte type	b5
generic 6 byte type	b6
generic 7 byte type	b7
generic 8 byte type	b8
...	e

Modifier	Encoding
unsigned	U
const	C
volatile	V
signed (only for characters!)	S

Types	Notation	Encoding
simple	Complex	7Complex
qualified	X::YY	Q21X2YY
pointer	*	P
__far pointer	* __far	PKF
__near pointer	* __near	PKN
reference	&	R
array	[10]	A10_
function	()	F
global operator	op ()	FG
pointer to member	S::*	M1S

Structs, classes and enumerations are encoded using the ‘simple’ notation:

```
void foo(struct MyStruct, class MyClass, enum MyEnum);
```

is encoded as:

```
foo__F8MyStruct7MyClass6MyEnum
```

Operator	Encoding
*	_ml
/	_dv
%	_md
+	_pl
-	_mi
>>	_rs
<<	_ls
==	_eq
!=	_ne
<	_lt
>	_gt

Operator	Encoding
<=	_le
>=	_ge
&	_ad
	_or
^	_er
&&	_aa
	_oo
!	_nt
~	_co
++	_pp
--	_mm
=	_as
->	_rf
+=	_apl
-=	_ami
*=	_aml
/=	_adv
%=	_amd
<<=	_als
>>=	_ars
&=	_aad
=	_aor
^=	_aer
,	_cm
->*	_rm
'?': (not supported yet)	_mn
'!': (not supported yet)	_mx
??: (not supported yet)	_cn

Operator	Encoding
() (function call)	__cl
[] (subscripting)	__vc
constructor	__ct
destructor	__dt
operator new()	__nw
operator delete()	__dl
operator T()	__op<signature of T>

Generating Compact Code

The Compiler tries whenever possible to generate compact and efficient code. But not everything is handled directly by the Compiler. With a little help from the programmer, it is possible to reach denser code. Some Compiler options, or using `__SHORT_SEG` segments (if available), help you to generate compact code.

Compiler Options

Using the following compiler options helps you to reduce the size of the code generated. Note that not all options may be available for your target.

-Or: Register Optimization

When accessing pointer fields, this option prevents the compiler from reloading the address of the pointer for each access. An index register holds the pointer value over statements where possible.

NOTE This option may not be available for all targets.

-Oi: Inline Functions

Use the inline keyword or the command line option `-Oi` for C++ functions. Defining a function before it is used helps the Compiler to inline it:

```
/* ok */                                /* better! */
void foo(void);                          void foo(void) {
void main(void) {                      // ...
    foo();                            }
}                                     void main(void) {
void foo(void) {                      foo();
    // ...                            }
}
```

This also helps the compiler to use a relative branch instruction instead an absolute.

__SHORT_SEG Segments

Variables allocated on the direct page (between 0 and 0xFF) are accessed using the direct addressing mode. You can instruct the Compiler that some variables are

allocated on the direct page if you define them in a `__SHORT_SEG` segment (not available for all targets).

Example:

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
unsigned int myvar3, myVar4.
```

In the previous example, ‘myVar1’ and ‘myVar2’ are both accessed using the direct addressing mode. Variables ‘myVar3’ and ‘myVar4’ are accessed using the extended addressing mode.

When some exported variables are defined in a `__SHORT_SEG` segment, the external declaration for these variables must also specify that they are allocated in a `__SHORT_SEG` segment. The External definition of the variable defined above looks like:

```
#pragma DATA_SEG __SHORT_SEG myShortSegment
extern unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
extern unsigned int myvar3, myVar4.
```

The segment must be placed on the direct page in the PRM file.

Example:

```
LINK test.abs
NAMES test.o startup.o ansi.lib END
SECTIONS
    Z_RAM = READ_WRITE 0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE 0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY 0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM           INTO MY_ROM;
    DEFAULT_RAM           INTO MY_RAM;
    _ZEROPAGE, myShortSegment INTO Z_RAM;
END
STACKSIZE 0x60
VECTOR 0 _Startup /* set reset vector on _Startup */
```

NOTE The linker is case sensitive. The segment name must be identical in the C and PRM file.

Defining IO Registers

The I/O Registers are usually based at address 0. In order to tell the compiler it must use direct addressing mode to access the IO registers, these registers are defined in a `__SHORT_SEG` section (if available) based at the specified address.

The IO register is defined in the C source file as follows:

Example:

```
typedef struct {
    unsigned char SCC1;
    unsigned char SCC2;
    unsigned char SCC3;
    unsigned char SCS1;
    unsigned char SCS2;
    unsigned char SCD;
    unsigned char SCBR;
} SCIstruct;
#pragma DATA_SEG __SHORT_SEG SCIREgs
SCIstruct SCI;
#pragma DATA_SEG DEFAULT
```

Then the segment must be placed at the appropriate address in the PRM file.

Example:

```
LINK test.abs
NAMES test.o startup.o ansi.lib END
SECTIONS
    SCI_RG = READ_WRITE 0x0013 TO 0x0019;
    Z_RAM = READ_WRITE 0x0080 TO 0x00FF;
    MY_RAM = READ_WRITE 0x0100 TO 0x01FF;
    MY_ROM = READ_ONLY 0xF000 TO 0xFEFF;
PLACEMENT
    DEFAULT_ROM           INTO MY_ROM;
    DEFAULT_RAM           INTO MY_RAM;
    _ZEROPAGE              INTO Z_RAM;
    SCIREgs                INTO SCI_RG;
END
STACKSIZE 0x60
VECTOR 0 _Startup /* set reset vector on _Startup */
```

NOTE	The linker is case sensitive. The segment name must be identical in the C and PRM file.
-------------	-----------------------------------------------------------------------------------------

Programming Guidelines

Following a few programming guidelines helps you reduce code size. Many things are optimized by the Compiler. However, if the programming style is very complex or if it forces the Compiler to perform special code sequences, code efficiency is not what you would expect from a typical optimization.

Constant Function at a Specific Address

Sometimes functions are placed at a specific address, but you do not have the sources or information regarding them. You just know that the function starts at address 0x1234 and you want to call it. Without having the definition of the function, you run into a linker error because you do not have the target function code. The solution is to use a constant function pointer:

```
void (*const fktPtr)(void) = (void(*)(void))0x1234;  
void main(void) {  
    fktPtr();  
}
```

This gives you efficient code and no linker errors. But you must ensure that the function at 0x1234 really exists.

Even a better way (without the need for a function pointer):

```
#define erase ((void(*)(void))(0xfc06))  
void main(void) {  
    erase(); /* call function at address 0xfc06 */  
}
```

HLI Assembly

Do not mix High Level Inline (HLI) Assembly. Using HLI assembly may affect the register trace of the compiler. The Compiler cannot touch HLI Assembly and thus it is out of range for any optimizations (except branch optimization, of course).

The Compiler in the worst case has to assume that everything has changed. It cannot hold variables into registers over HLI statements. Normally it is better to place special HLI code sequences into separate functions. However, there is the drawback of an additional call/return. Placing HLI instructions into separate functions (and module) simplify porting the software to another target.

Example (not recommended):

```
void foo(void) {
    /* some local variable declarations */
    /* some C/C++ statements */
    asm {
        /* some HLI statements */
    }
    /* maybe other C/C++ statements */
}
```

Example (recommended):

```
/* hardware.c */
void special_hli(void) {
    asm {
        /* some HLI statements */
    }
}

/* foo.c */
void foo(void) {
    /* some local variable declarations */
    /* some C/C++ statements */
    special_hli();
    /* maybe other C/C++ statements */
}
```

Post/Pre Operators in Complex Expressions

Writing a complex program results in complex code. In general it is the job of the compiler to optimize complex functions. Some rules may help the compiler to generate efficient code.

If your target does not support powerful postincrement/postdecrement and preincrement/predecrement instructions, it is not recommended to use the ‘`++`’ and ‘`--`’ operator in complex expressions. Especially postincrement/postdecrement may result in additional code:

```
a[i++] = b[--j];
```

Write above statement as:

```
i--; a[i] = b[j]; j--;
```

Using it in simple expressions as:

```
i++;
```

Avoid assignments in parameter passing or side effects (as ‘`++`’ and ‘`--`’). The evaluation order of parameters is undefined (ANSI C standard 6.3.2.2) and may vary from Compiler to Compiler, and even from one release to another:

Example:

```
i = 3;  
foo(i++, --i);
```

In the above example, `foo()` is called either with ‘`foo(3,3)`’ or with ‘`foo(2,2)`’.

Boolean Types

In C, the boolean type of an expression is an ‘`int`’. A variable or expression evaluating to ‘0’ (zero) is FALSE and everything else ($\neq 0$) is TRUE. Instead of using an ‘`int`’ (usually 16 or 32 bit), it may be better to use an 8-bit type to hold a boolean result. For ANSI-C compliance, the basic boolean types are declared in ‘`stdtypes.h`’:

```
typedef int Bool;  
#define TRUE 1  
#define FALSE 0
```

Using

```
typedef Byte Bool_8;
```

from ‘`stdtypes.h`’ (‘`Byte`’ is an unsigned 8bit data type also declared in ‘`stdtypes.h`’) reduces memory usage and improves code density.

Printf/Scanf

You can reduce the `printf`/`scanf` in the ANSI library if no floating point support (%f) is used. Refer to the ANSI library reference and the `printf.c`/`scanf.c` in your library for details on how to save code (not using float/doubles in `printf` may result in half the code).

Bitfields

Using bitfields to save memory may be a bad idea as bitfields produce a lot of additional code. For ANSI-C compliance, bitfields have a type of ‘`signed int`’, thus a bitfield of size 1 is either ‘-1’ or ‘0’. This could force the compiler to ‘sign extend’ operations:

```

struct {
    int b:0; /* -1 or 0 */
} B;

int i = B.b; /* load the bit, sign extend it to -1 or 0 */

```

Sign extensions are normally time and code inefficient operations.

Struct Returns

ANSI-C/C++ allows functions that return a struct, unless the struct is returned in a single register. Returning a struct can force the Compiler to produce a lengthy code:

```

struct S foo(void) {
    /* ... */
    return s; // (4)
}

void main(void) {
    struct S s;
    /* ... */
    s = foo(); // (1), (2), (3)
    /* ... */
}

```

Normally the compiler has first to allocate space on the stack for the return value (1) and then to call the function (2). In the callee ‘foo’ during the return sequence, the Compiler has to copy the return value (4, struct copy).

Depending on the size of the struct, this may be done inline or with a special memory copy routine (strcpy). After return, the caller ‘main’ has to copy the result back into ‘s’. Depending on the Compiler/Target, it is possible to optimize some sequences (avoiding some copy operations). However, returning a struct by value may use a lot of execution time and means a lot of code and stack usage.

A better way is pass only a pointer to the callee for the return value:

```

void foo(struct S *sp) {
    /* ... */
    *sp = s; // (4)
}

void main(void) {
    S s;
    /* ... */
    foo(&s); // (2)
    /* ... */
}

```

With the above example, the Compiler just has to pass the destination address and to call ‘foo’ (2). On the callee side, the callee copies the result indirectly into the destination (4). This approach reduces stack usage, avoids copying structs, and results in denser code. Note that the Compiler may also inline the above sequence (if supported). But for rare cases the above sequence may not be exactly the same as returning the struct by value (e.g. if the destination struct is modified in the callee).

Local Variables

Using local variables instead of global variable results in better manageability of the application as side effects are reduced or totally avoided. Using local variables/parameters reduces global memory usage but increases stack usage.

Stack access capabilities of the target influences the code quality. Depending on the target capabilities, access to local variables may be very inefficient. The reason being that no dedicated stack pointer (another address register has to be used instead, thus it might not be used for other values) or access to local variables is inefficient due the target architecture (limited offsets, only few addressing modes).

Allocating a huge amount of local variables may be inefficient because the Compiler has to generate a complex sequence to allocate the stack frame in the beginning of the function and to deallocate them in the exit part:

```
void foo(void) {  
    /* huge amount of local variables: allocate space! */  
    /* ... */  
    /* deallocate huge amount of local variables */  
}
```

If the target provides special entry/exit instructions for such cases, allocation of many local variables is not a problem. A solution is to use global or static local variables. This deteriorates maintainability and also may waste global address space.

The Compiler may offer an option to overlap parameter/local variables using a technique called ‘overlapping’. Local variables/parameters are allocated as global ones. The linker overlaps them depending on their use. For targets with limited stack (e.g. no stack addressing capabilities), this often is the only solution. However this solution makes your code non-reentrant (no recursion is allowed).

Parameter Passing

Avoid parameters which exceed the data passed through registers (see Back End).

Unsigned Data Types

Using unsigned data types is acceptable as signed operations are much more complex than unsigned ones (e.g. shifts, divisions and bitfield operations). But it is a bad idea to use unsigned types just because a value is always larger or equal to zero, and because the type can hold a larger positive number.

Inlining/Macros

abs() and abs()

Use the corresponding macro M_ABS defined in stdlib.h instead of calling abs() and abs() in the stdlib:

```
/* extract
/* macro definitions of abs() and labs() */
#define M_ABS(j) (((j) >= 0) ? (j) : -(j))
extern int      abs    (int j);
extern long int labs   (long int j);
```

But be careful as M_ABS() is a macro,

```
i = M_ABS(j++);
```

and is not the same as:

```
i = abs(j++);
```

memcpy() and memcpy2()

ANSI-C requires that the memcpy() library function in ‘strings.h’ returns a pointer of the destination and handles, and is able to also handle a count of zero:

```
/* extract of string.h */
extern void*
memcpy(void *dest, const void *source, size_t count);

extern void
memcpy2(void *dest, const void* source, size_t count);
/* this function doesn't return dest and assumes count>0 */

/* extract of string.c */
void*
memcpy(void *dest, const void *source, size_t count) {
    uchar *sd = dest;
    uchar *ss = source;
```

```
    while (count--)
        *sd++ = *ss++;

    return (dest);
}
```

If the function does not have to return the destination and it does have to handle a count of zero, the memcpy below is much more simpler and faster:

```
/* extract of string.c */
void
memcpy2(void *dest, const void* source, size_t count) {
    /* this func does not return dest and assumes count > 0 */
    do {
        *((uchar *)dest)++ = *((uchar*)source)++;
    } while(count--);
}
```

Replacing calls to memcpy() with calls to memcpy2() saves runtime and code size.

Data Types

Do not use larger data types than necessary. Use IEEE32 floating point format both for float and doubles if possible. Set the enum type to a smaller type than ‘int’ using the -T option. Avoid data types larger than registers.

Short Segments

Whenever possible and available (not all targets support it), place frequently used global variables into a DIRECT or __SHORT_SEG segment using

```
#pragma DATA_SEG __SHORT_SEG MySeg
```

Qualifiers

Use the ‘const’ qualifier to help the compiler. The ‘const’ objects are placed into ROM for the HIWARE object file format if the command line option -Cc is given.

Freescale XGATE Back End

The Back End is the target-dependent part of a Compiler, containing the code generator. This section discusses the technical details of the Back End for the XGATE family.

Non-ANSI Keywords

The following table gives an overview about the supported non-ANSI keywords:

Keyword	Data Pointer	Supported for Function Pointer	Function
<code>__far</code>	no	no	no
<code>__near</code>	no	no	no
<code>interrupt</code>	no	no	yes

Data Types

This section describes how the basic types of ANSI-C are implemented by the XGATE Back End.

Scalar Types

All basic types may be changed with the [-T](#) option. Note that all scalar types (except char) have no signed/unsigned qualifier, and are considered signed by default, e.g. ‘int’ is the same as ‘signed int’.

The sizes of the simple types are given by the table below together with the possible formats using the [-T](#) option:

Type	Default Format	Default Value Range		Formats Available With Option -T
		Min	Max	
char (unsigned)	8bit	-0	255	8bit, 16bit, 32bit
signed char	8bit	-128	127	8bit, 16bit, 32bit
unsigned char	8bit	0	255	8bit, 16bit, 32bit

Type	Default Format	Default Value Range		Formats Available With Option -T
		Min	Max	
signed short	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned short	16bit	0	65535	8bit, 16bit, 32bit
enum (signed)	16bit	-32768	32767	8bit, 16bit, 32bit
signed int	16bit	-32768	32767	8bit, 16bit, 32bit
unsigned int	16bit	0	65535	8bit, 16bit, 32bit
signed long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long	32bit	0	4294967295	8bit, 16bit, 32bit
signed long long	32bit	-2147483648	2147483647	8bit, 16bit, 32bit
unsigned long long	32bit	0	4294967295	8bit, 16bit, 32bit

NOTE Plain type `char` is signed. This default is changed with the [-T](#) option.

Floating Point Types

The XGATE compiler does support IEEE32 floating point calculations. The compiler uses IEEE32 format for both `float` and `double` types.

The option [-T](#) may be used to change the default format of `float`/`double`.

Type	Default Format	Default Value Range		Formats Available With Option -T
		Min	Max	
<code>float</code>	IEEE32	-1.17549435E-38F	3.402823466E+38F	IEEE32
<code>double</code>	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32
<code>long double</code>	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32
<code>long long double</code>	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32

Pointer Types and Function Pointers

The size of pointer types depends on the memory model selected. The following table gives an overview.

Type	Example	Size
default data pointer	char*	2 bytes
default function pointer	void (*)(void)	2 bytes

Structured Types, Alignment

Local variables are allocated on the stack (which grows downwards). The most significant part of a simple variable always is stored at the low memory address.

Bit Fields

The maximum width of bit fields is 32 bits. The allocation unit is one byte. The Compiler uses words only if a bit field is wider than eight bits. Allocation order is from the least significant bit up to the most significant bit in the order of declaration.

Register Usage

The Compiler uses all registers of the XGATE::

Register	Interrupt Function	Normal Function
R1	First (and only) argument	callee saved. Entry code saves R1 if it is modified
R2	Scratch	first argument and first part of return value scratch register if not used for argument/return
R3	Scratch	second argument and second part of return value scratch register if not used for argument/return
R4	Scratch	third argument scratch register if not used for arguments
R5	Scratch	callee saved. Entry code saves R1 if it is modified

Register	Interrupt Function	Normal Function
R6	function call register Calls are encoded with JAL R6.	function call register Calls are encoded with JAL R6.
R7	Stack pointer. Loaded at start of interrupt function. Initial stack value defined with option _Cstv .	stack pointer. Decremented to reserve space on stack.

Call Protocol and Calling Conventions

Argument Passing

The Pascal calling convention is used for functions with a fixed number of parameters: The caller pushes the arguments from left to right. After the call, the caller removes the parameters from the stack.

For functions with a variable number of parameters, the caller pushes the arguments from right to left.

For functions with a fixed number of arguments, the last 3 arguments are passed in the registers R2, R3, R4 if the argument types are 16 bit or smaller.

The following table gives an overview of the registers used for argument passing.

size of last arguments	Type Example	Register
2 (or 1)	int	R2
2 (or 1), 2 (or 1)	char, void*	R3, R2
2 (or 1), 2 (or 1), 2 (or 1)	int, int, void(*)(void)	R4, R3, R2
4	long	R2:R3
4, 2 (or 1)	long, int	R3:R4, R2
2 (or 1), long	int, long	R4, R2:R3

All previous parameters are passed on the stack.

Return Values

Function results are returned in registers, except if the function returns a result larger than one word (see below). Depending on the return type, different registers are used.

Size of Return Value	Type Example	Register
1 byte	char	R2
2 bytes	int	R2
4 bytes	long	R2:R3

Returning Large Results

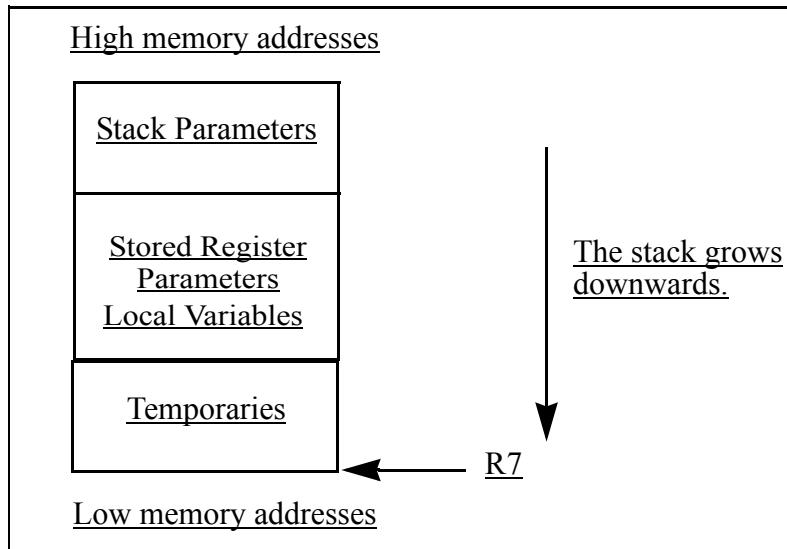
Functions returning a result larger than two words are called with an additional parameter. This parameter is the address where the result should get copied to.

Stack Frames

Functions have a stack frame containing all their local data. The Compiler uses the stack pointer as the base address for accessing local data.

If one of the pragmas NO_ENTRY, NO_EXIT or NO_FRAME is active, the Compiler does not generate code to set up a stack frame for this function. In this case the function must have neither local variables nor parameters passed on the stack.

The figure below shows the stack frame of a normal function, i.e. compiled with above pragmas inactive.



Entry Code

Normal *entry code* is a sequence of instructions reserving space for local variables and writing eventually the register parameter to the stack. The actual entry code generated does depend on many properties, so here just one example:

The following code:

```
void f(int*, int, int*, int*);  
void test(int a, int b, int c, int d) {  
    int local;  
    f(&a, b, &d, &local);  
}
```

Generates the following entry code:

```
STW R6,(-SP) ; save return address  
STW R2,(-SP) ; save d argument because its adr is taken  
SUBL R7,#2 ; reserve space for local because adr is taken
```

In this example, the unused argument c does not generate any code. The argument b is directly kept in a register and therefore no entry code is generated. The a argument is already on the stack, so no entry code for it is generated as well.

Exit Code

Exit code removes local variables from the stack before returning to the caller. It does reload the return address (if it was saved on the stack) and finally returns with a "JAL R6".

For the example in the Entry Code section above, the following exit code is generated:

```

ADDL R7,#4      ; release stack space of d and index at once.
LDW  R6,(SP+)   ; reload return address
JAL  R6         ; jump back to caller

```

For interrupt functions (with the pragma TRAP_PROC or the interrupt keyword), different entry and exit code is generated.

With the option -Cstv=0xe000 (meaning the stack ends at 0xDFFF), the code:

```

struct SciArguments {
    struct sciDescr* ptr;
    const char* send_buf;
    int bufsize;
    int bufpos;
};

void SendByte(char);
void interrupt SciInt(struct SciArguments* restrict hand) {
    if (hand->bufsize != hand->bufpos) {
        SendByte(hand->send_buf[hand->bufpos]);
        hand->bufpos++;
    }
}

```

generates the following:

```

LDL      R7,#0      ; avoided with option -CsIni0
LDH      R7,#224     ; load $E000 in R7
>      if (hand->bufsize != hand->bufpos) {
        LDW      R2,(R1,#4)  ; load bufsize
        LDW      R5,(R1,#6)  ; load bufpos
        CMP      R2,R5
        BEQ      L1A
>      SendByte(hand->send_buf[hand->bufpos]);
        LDW      R2,(R1,#2)  ; load send_buf
        LDB      R2,(R2,R5)  ; load send_buf[bufpos]
        LDL      R6,%XGATE_8(SendByte)

```

```
        ORH      R6 , #%%XGATE_8_H(SendByte)
        JAL      R6          ; call SendByte
>      hand->bufpos++;
        ADDL    R5 , #1       ; store incremented
        STW     R5 , (R1 , #6) ; bufpos
L1A:
        RTS
```

For interrupt function no registers are saved. R7 is loaded with the initial stack address (of the option -Cstv is present).

The exit code consists of a single RTS (Return From Scheduler).

Pragmas

The Compiler provides a couple of pragmas that control the allocation of stack frames and the generation of entry and exit code.

TRAP_PROC

The procedure terminates with an RTS instruction instead of an "JAL R6". The same effect can be achieved with the interrupt keyword.

NO_ENTRY

Omits generation of procedure entry code.

NO_EXIT

Does not generate procedure exit code. It's the programmer's responsibility that the function does return somehow!

NO_FRAME

No stack frame is set up, but the Compiler generates an JAL R6 (or RTS, if pragma TRAP_PROC/interrupt keyword is active).

Interrupt Functions

Interrupt procedures are quite different from other procedures.

- The function returns with a RTS.

- no registers must be saved.
- interrupt functions can either have no arguments or exactly one with either 8 or 16 bit, This argument is passed in R1 (and not in R2 as it would be for other functions).
- interrupt functions must not clean up the stack,
- interrupt functions must load the R7 register with the initial stack pointer value (see option [-Cstv](#)).

#pragma TRAP_PROC

Which page registers are saved is determined by pragma TRAP_PROC. The syntax of this pragma is

```
#pragma TRAP_PROC
```

Interrupt Vector Table Allocation

The vector table has to be setup with a normal C (or assembly) code. The interrupt number feature for the interrupt vector is not supported for the XGATE (as it is for the HC12).

Instead an array of vectors has to be allocated and initialized with the address of the handlers and with their initial thread argument.

Intrinsics

The compiler does support some intrinsics. Intrinsics look like ANSI-C functions, but instead of calling a function, use them does directly generate some special assembly code. The following lists the supported intrinsics for the XGATE processor.

Semaphore handling

The semaphore handling intrinsics are used to provide access to the hardware semaphores provided to control the concurrent access to a resource from the HCS12X and the XGATE from the XGATE side.

Example:

```
while (!_ssem(1)) { }
..use resource protected by semaphore
_csem(1);
```

_ssem: set XGATE semaphore

The `_ssem` intrinsic is implemented with the SSEM instruction.

Syntax:

```
int _ssem(unsigned int sem);
```

_csem: clear XGATE semaphore

The `_csem` intrinsic is implemented with the CSEM instruction.

Syntax:

```
void _csem(unsigned int sem);
```

Mathematical intrinsics

The mathematical intrinsics are used to perform some calculations especially efficient on the XGATE architecture.

Example:

```
int bits= ...;
int firstOne= _bffo(bits);
```

_par: calculate parity

The `_par` intrinsic is implemented with the PAR instruction.

Syntax:

```
unsigned int _par(unsigned int);
```

_bffo: find first one

The `_bffo` intrinsic implemented with the BFFO instruction.

Syntax:

```
int _bffo(unsigned int);
```

_rol: rotate left

The `_rol` intrinsic implemented with the ROL instruction.

Syntax:

```
unsigned int _rol(unsigned int val, unsigned int cnt);
```

_ror: rotate right

The `_ror` intrinsic implemented with the ROR instruction.

Syntax:

```
unsigned int _ror(unsigned int val, unsigned int cnt);
```

Flag handling

The flag handling intrinsics are used to check hardware supported flags not otherwise accessible from C.

Example:

```
int bits= ...;  
int firstOne= _bffo(bits);  
if (_carry()) {...}
```

_carry: check carry flag

The `_carry` intrinsic is usually used in conditions. If so, it is implemented with either a BCC or a BCS instructions. When the `_carry` intrinsic is used in a general expression context, then it is implemented with a ADC RX,R0,R0 instruction.

Syntax:

```
unsigned int _carry(void);
```

_ovfl: check overflow flag

The `_ovfl` intrinsic must be used in a condition. It is implemented with either a BVC or a BVS instruction.

Syntax:

```
unsigned int _ovfl(void);
```

Interrupt signaling

The interrupt signaling intrinsics are used by the XGATE to raise an interrupt for the HCS12X.

This might indicate that some data for the HCS12X is ready or any other topic which should be handled by the HCS12X.

Example:

```
if (receiveError) {  
    _sif(HCS12_RECEIVE_ERROR_INTERRUPT);  
}
```

_sif: signal current interrupt flag

The sif intrinsic does forwards the interrupt just being handled by the XGATE to the HCS12X.

It is implemented with the SIF instruction.

Syntax:

```
void _sif(void);
```

_sif1: signal interrupt flag with channel number argument

The sif1 intrinsic does raise a specified interrupt for the HCS12X.

It is implemented by the SIF instruction with a argument.

Syntax:

```
void _sif1(int chan);
```

Segmentation

The Linker memory space may be partitioned into several segments. The Compiler allows attributing a certain segment name to certain global variables or functions which then are allocated into that segment by the Linker. Where that segment actually lies is determined by an entry in the Linker parameter file.

There are two basic types of segments, code and data segments, each with a matching pragma:

```
#pragma CODE_SEG <name>
#pragma DATA_SEG <name>
```

Both are valid until the next pragma of the same kind is encountered. If no segment is specified, the Compiler assumes two default segments named `DEFAULT_ROM` (the default code segment) and `DEFAULT_RAM` (the default data segment). To explicitly make these default segments the current ones, use the segment name `DEFAULT`:

```
#pragma CODE_SEG DEFAULT
#pragma DATA_SEG DEFAULT
```

Optimizations

The Compiler applies a variety of code improving techniques commonly defined as “optimization”. This section gives a short overview about the most important optimizations.

Lazy Instruction Selection

Lazy instruction selection is a very simple optimization that replaces certain instructions by shorter and/or faster equivalents. Examples are the use of `TSTA` instead of `CMPA #0` or using `COMB` instead of `EORB #0xFF`.

Branch Optimizations

The Compiler uses branch instructions with short offsets whenever possible. Additionally, other optimizations for branches are also available.

Constant Folding

Constant folding options only affect constant folding over statements. The constant folding inside of expressions is always done.

Volatile Objects

The Compiler does not do register tracing on volatile objects. Accesses to volatile objects are not eliminated. It also doesn't change word operations to byte operations on volatile objects as it does for other memory accesses.

Programming Hints

The XGATE is an 8/16-bit processor not designed with high-level languages in mind. You must observe certain points in order to allow the Compiler to generate reasonably efficient code. The following list provides an idea of what is “good” programming from the processor’s point of view.

- Use the `restrict` keyword as hint for the pointer to thread function argument descriptors.
- The XGATE core is a 16 bit RISC architecture and therefore it is sometimes better to use a 16 bit `int` than to use a 8 bit (`unsigned`) `char`. Especially compares are less expensive because `char` compares do often need to cut the register to 8 bit first.

Using `unsigned` types instead of `signed` types is better in the following cases:

- Implicit or explicit extensions from `char` to `int` or from `int` to `long`.
- Use types `long`, `float` or `double` only when absolutely necessary. They produce a lot of code!
- Avoid stack frames larger than 32 bytes. The stack frame includes the parameters, local variables and usually some additional bytes for temporary values.

High Level Inline Assembler for Freescale XGATE

The HLI (High Level Inline) Assembler provides a means to make full use of the properties of the target processor right within a C program. There is no need to write a separate assembly file, assemble it and later bind it with the rest of the application written in ANSI-C/C++ with the inline assembler. The Compiler does all that work for you. For further information, please refer to the XGATE Reference Manual.

Syntax

Inline assembly statements can appear anywhere a C statement can appear (an `asm` statement must be inside a C function). Inline assembly statements take one of two forms, shown in various configurations:

```
"asm" <Assembly Instruction> ";" [ /* Comment */ ]  
"asm" <Assembly Instruction> ";" [ // Comment ]
```

or

```
"asm" "{"  
{ <Assembly Instruction> [ ";" Comment ] "\n" }  
"}"
```

or

```
"asm" " (" <Assembly Instruction> ";" [ /* Comment */ ]  
" )" ";"
```

or

```
"asm" [ " ( " ] <string Assembly instruction > [ " ) " ] [ ";" ]  
with <string Assembly instruction >  
= <Assembly Instruction> [ ";" <Assembly instruction> ]
```

or

```
#asm  
<Assembly Instruction> [ ";" Comment ] "\n"  
#endasm"
```

If you use the first form, multiple `asm` statements are contained on one line and comments are delimited like regular C or C++ comments. If you use the second form, one to several assembly instructions are contained within the `asm` block, but only one assembly instruction per line is possible and the semicolon starts an assembly comment.

Mixing HLI Assembly and HLL

Mixing High Level Inline (HLI) Assembly with a High Level Language (HLL, e.g. C or C++) requires special attention. The Compiler does care about used/modified registers in HLI Assembly, thus you do not have save/restore registers which are used in HLI. It is recommended to place complex HLI Assembly code, or HLI Assembly code modifying any registers, into separate functions.

Example:

```
void foo(void) {
    /* some C statements */
    p->v = 1;
    asm {
        /* some HLI statements destroying registers */
    }
    /* some C statements */
    p->v = 2;
}
```

In the above sequence, the Compiler holds the value of p in a register. The compiler will correctly reload p if necessary.

Example

A simple example illustrates the use of the HLI-Assembler. Assume the following:

- `from` points to some memory area
- `to` points to some other, non-overlapping memory area.

Then we can write a simple string copying function in assembly language as follows:

```
#pragma NO_ENTRY
char* strcpy(char* to, const char* from) {
    __asm {
        STW      R6,(R0,-R7)
        MOV      R6, R2
        Loop:
        LDB      R4,(R0,R6+)
        STB      R4,(R0,R3+)
        CMP      R4,R0
        BNE      Loop
        LDW      R6,(R0, R7+)
        JAL      R6
    }
}
```

NOTE	If #pragma NO_ENTRY is not set, the Compiler takes care of entry and exit code. You do not have to worry about setting up a stack frame.
-------------	------------------------------------------------------------------------------------------------------------------------------------------

C Macros

The C macros are expanded inside of inline assembler code as they are expanded in C. One special point to note is the syntax of asm directive generated by macros. As macros always expand to one single line, only the first form of the asm keyword is used in macros:

```
asm NOP;
```

E.g.

```
#define SPACE_OK { asm NOP; asm NOP; }
```

Using the second form is illegal:

```
#define NOT_OK { asm { \
    NOP; \
    NOP; \
}}
```

The macro, NOT_OK, is expanded by the preprocessor to one single line, which is then incorrectly translated because every assembly instruction must be explicitly terminated by a newline. Use the [pragma NO_STRING_CONSTR to build immediates by using #](#) inside macros.

Special Features

Caller/Callee Saved Registers

The compiler does assume that R1 and R5 do remain valid across function calls. Therefore assembly functions have to make sure this condition holds as well if they are called from C code.

Reserved Words

The inline assembler knows a couple of reserved words, which must not collide with user defined identifiers such as variable names. These reserved words are:

- All opcodes (MOV, NOP, ...)

- All register names (R1, R2, R3, R4, R5, R6, R7)
- The fixup identifiers:

Name	Address Kind	Description
%LOGICAL_8	Logical Address	LSB. Bits 0..7 of logical address.
%LOGICAL_8_H	Logical Address	2nd Byte. Bits 8..15 of logical address.
%LOGICAL_16	Logical Address	Bits 0..15 of logical address.
%LOGICAL_32	Logical Address	Bits 0..31 of logical address. (Bits 24..31 are always 0)
%GLOBAL_8	Global Address	LSB. Bits 0..7 of global address.
%GLOBAL_8_H	Global Address	2nd Byte. Bits 8..15 of global address.
%GLOBAL_16	Global Address	Bits 0..15 of global address.
%GLOBAL_32	Global Address	Bits 0..31 of global address. (Bits 23..31 are always 0)
%XGATE_8	XGATE Address	LSB. Bits 0..7 of xgate address. Causes link time error if address is not in RAM/Register area.
%XGATE_8_H	XGATE Address	2nd Byte. Bits 8..15 of xgate address. Causes link time error if address is not in RAM/Register area.
%XGATE_16	XGATE Address	Bits 0..15 of xgate address. Causes link time error if address is not in RAM/Register area.

For these reserved words, the inline assembler is *not* case sensitive, i.e. LDAB is the same as ldab or even LdAb. For all other identifiers (labels, variable names and so on) the inline assembler is case sensitive.

The following example shows how to load the address of a function in order to call it. Note the syntax of the fixup specifications.

```
void fun(void);
#pragma NO_EXIT
#pragma NO_FRAME
#pragma NO_ENTRY
void test(void) {
    __asm {
```

```
        STW      R6 , (R0 , -R7)
        LDL      R6 , #%XGATE_8(fun)
        ORH      R6 , #%XGATE_8_H(fun)
        JAL      R6
        LDW      R6 , (R0 , R7+)
        JAL      R6
    }
}
```

Pseudo-Opcodes

The inline assembler provides some pseudo opcodes to put constant bytes into the instruction stream. These are:

```
DC.B 1      ; Byte constant 1
DC.B 0      ; Byte constant 0
DC.W 12     ; Word constant 12
DC.L 20,23  ; Longword constants
```

Accessing Variables

The inline assembler allows accessing local and global variables declared in C by using their name in the instruction. For global variable names, use the correct fixup specification (usually %XGATE_8 for the low byte and %XGATE_8_H for the high byte part).

Constant Expressions

Constant expressions may be used anywhere an *IMMEDIATE* value is expected. The HLI supports the same operators as in ANSI-C code. The syntax of numbers is the same as in ANSI-C.

Using the Compiler

High Level Inline Assembler for Freescale XGATE

Messages

This chapter describes the messages produced by the application. Because of the large amount of different messages, not all of them may be in this document as of this release.

Message Kinds

The Compiler generates five kinds of messages:

- INFORMATION
A message is printed and the compilation continues. Information messages indicate actions taken by the application.
- WARNING
A message is printed and processing continues. Warning messages indicate possible programming errors.
- ERROR
A message is printed and processing is stopped. Error messages indicate illegal use of the language.
- FATAL
A message is printed and processing is aborted. A fatal message indicates a severe error which will stop processing.
- DISABLE
The message has been disabled. No message is issued and processing continues as if there was no such message. The application ignores this kind of message.

Message Details

If the application prints out a message, the message contains a message code and a four to five digit number. This number may be used to search for the indicated message.

Messages

Message List

The following message codes are supported:

- “A” for Assemblers
- “B” for Burner
- “C” for Compilers
- “D” for Decoder
- “L” for Linker
- “LM” for Libmaker
- “M” for Maker

All messages generated by the application are documented in increasing numerical order for easy and fast retrieval. Each message also has a description and if available, a short example with a possible solution or tips to fix a problem.

For each message the type of the message is also noted, e.g. [ERROR] indicates that the message is an error message.

[DISABLE, INFORMATION, WARNING, ERROR]

This indicates that the message is a warning message by default. You can change the message either to DISABLE, INFORMATION or ERROR.

After the message kind, there may be an additional entry telling for which language the message is generated:

- C++: Message is generated for C++
- M2: Message is generated for Modula-2

Message List

The following pages describe all messages.

All message numbers below 10000 are common to all compilers. Not every compiler emits all of them. For example, many compilers support any type of struct return. Such compilers will never emit the message “C2500: Expected: No support of class/struct return type”.

C1: Unknown message occurred

[FATAL]

Description

The application tried to emit a message which was not defined. This is a internal error which should not occur. Please report any occurrences to you distributor.

Tips

Try to find out the and avoid the reason for the unknown message.

C2: Message overflow, skipping <kind> messages

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The application did show the number of messages of the specific kind as controlled with the options [-WmsgNi](#), [-WmsgNw](#) and [-WmsgNe](#). Further options of this kind are not displayed.

Tips

Use the options [-WmsgNi](#), [-WmsgNw](#) and [-WmsgNe](#) to change the number of messages.

C50: Input file '<file>' not found

[FATAL]

Description

The Application was not able to find a file needed for processing.

Tips

Check if the file really exists. Check if you are using a file name containing spaces (in this case you have to quote it).

C51: Cannot open statistic log file <file>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

It was not possible to open a statistic output file, therefore no statistics are generated.

Note: Not all tools support statistic log files. Even if a tool does not support it, the message still exists, but is never issued in this case.

C52: Error in command line <cmd>

[FATAL]

Description

In case there is an error while processing the command line, this message is issued.

C64: Line Continuation occurred in <FileName>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In any environment file, the character '\' at the end of a line is taken as line continuation. This line and the next one are handled as one line only. Because the path separation character of MS-DOS is also '\', paths are often incorrectly

written ending with '\'. Instead use a '.' after the last '\' to not finish a line with '\' unless you really want a line continuation.

Example

Current Default.env:

```
...
LIBPATH=c:\metrowerks\lib\
OBJPATH=c:\metrowerks\work
...
Is taken identical as
...
LIBPATH=c:\metrowerks\libOBJPATH=c:\metrowerks\work
...
```

Tips

To fix it, append a '.' behind the '\'

```
...
LIBPATH=c:\metrowerks\lib\.
OBJPATH=c:\metrowerks\work
...
```

Note

Because this information occurs during the initialization phase of the application, the message prefix might not occur in the error message. So it might occur as "64: Line Continuation occurred in <FileName>".

C65: Environment macro expansion message '<description>' for <variablename>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

During a environment variable macro substitution an problem did occur. Possible causes are that the named macro did not exist or some length limitation was reached. Also recursive macros may cause this message.

Example

Current variables:

```
...
LIBPATH=${LIBPATH}
...
```

Tips

Check the definition of the environment variable.

C66: Search path <Name> does not exist

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The tool did look for a file which was not found. During the failed search for the file, a non existing path was encountered.

Tips

Check the spelling of your paths. Update the paths when moving a project. Use relative paths.

C1000: Illegal identifier list in declaration

[ERROR]

Description

A function prototype declaration had formal parameter names, but no types were provided for the parameters.

Example

```
int f(i);
```

Tips

Declare the types for the parameters.

C1001: Multiple const declaration makes no sense

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The "constant" qualifier was used more than once for the same variable.

Example

```
const const int i;
```

Tips

Constant variables need only one "constant" qualifier.

See also

Qualifiers

C1002: Multiple volatile declaration makes no sense

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The "volatile" qualifier was used more than once for the same variable.

Example

```
volatile volatile int i;
```

Tips

Volatile variables need only one "volatile" qualifier.

C1003: Illegal combination of qualifiers

[ERROR]

Description

The combination of qualifiers used in this declaration is illegal.

Example

```
int *far near p;
```

Tips

Remove the illegal qualifiers.

C1004: Redefinition of storage class

[ERROR]

Description

A declaration contains more than one storage class.

Example

```
static static int i;
```

Tips

Declare only one storage class per item.

C1005: Illegal storage class

[ERROR]

Description

A declaration contains an illegal storage class.

Example

```
auto int i; // 'auto' illegal for global variables
```

Tips

Apply a correct combination of storage classes.

See also

Storage Classes

C1006: Illegal storage class

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A declaration contains an illegal storage class. This message is used for storage classes which makes no sense and are ignored (e.g. using ‘register’ for a global variable).

Example

```
register int i; //'register' for global variables
```

Tips

Apply a correct combination of storage classes.

C1007: Type specifier mismatch

[ERROR]

Description

The type declaration is wrong.

Example

```
int float i;
```

Tips

Do not use an illegal type chain.

C1008: Typedef name expected

[ERROR]

Description

A variable or a structure field has to be either one of the standard types (char, int, short, float, ...) or a type declared with a `typedef` directive.

Example

```
struct A {  
    type j; /* type is not known */  
} A;
```

Tips

Use a `typedef`-name for the object declaration.

C1009: Invalid redeclaration

[ERROR]

Description

Classes, structures and unions may be declared only once. Function redeclarations must have the same parameters and return values as the original declaration. In C++, data objects cannot be redeclared (except with the "extern" specifier).

Example

```
struct A {  
    int i;  
};  
  
struct A { // error  
    int i;  
};
```

Tips

Avoid redeclaration, e.g. guarding include files with '#ifndef'.

C1010: Illegal enum redeclaration

[ERROR]

Description

An enumeration has been declared twice.

Example

```
enum A {  
    B  
};  
  
enum A { //error  
    B  
};
```

Tips

Enums have to be declared only once.

C1012: Illegal local function definition

[ERROR], C++

Description

Non-standard error!

Example

```
void main() {
    struct A {
        void f() {}
    };
}
```

Tips

The function definitions must always be in the global scope.

```
void main(void) {
    struct A {
        void f();
    };
}

void A::f(void) {
    // function definition in global scope
}
```

C1013: Old style declaration

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler has detected an old style declaration. Old style declarations are common in old non-ANSI sources, however they are accepted. With old style declarations, only the names are in the parameter list and the names and types are declared afterwards.

Example

```
foo(a, b)
    int a, long b;
{
    ...
}
```

Tips

Remove such old style declarations from your application:

```
void foo(int a, long b) {  
    ...  
}
```

C1014: Integral type expected or enum value out of range

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A non-integral value was assigned to a member of an enum or the enumeration value does not fit into the size specified for the enum (in ANSI-C the enumeration type is int).

Example

```
enum E {  
    F="Hello"  
};
```

Tips

Enum-members may only get int-values.

C1015: Type is being defined

[ERROR]

Description

The given class or structure was declared as a member of itself. Recursive definition of classes and structures are not allowed.

Example

```
struct A {  
    A a;  
};
```

Tips

Use a pointer to the class being defined instead of the class itself.

C1016: Parameter redeclaration not permitted

[ERROR]

Description

A parameter object was declared with the same name as another parameter object of the same function.

Example

```
void f(int i, int i);
```

Tips

Choose another name for the parameter object with the already used name.

C1017: Empty declaration

[ERROR]

Description

A declaration cannot be empty.

Example

```
int;
```

Tips

There must be a name for an object.

C1018: Illegal type composition

[ERROR]

Description

The type was composed with an illegal combination.

A typical example is

```
extern struct A dummy[ ];
```

An array type describes a contiguously allocated non-empty set of objects with a particular member object type, called the element type. Since object types do not include incomplete types, an array of incomplete type cannot be constructed.

Example

```
void v[ 2];
```

Tips

Type compositions must not contain illegal combinations.

C1019: Incompatible type to previous declaration

[ERROR]

Description

The specified identifier was already declared

Example

```
int i;  
int i();
```

Tips

Choose another name for the identifier of the second object.

C1020: Incompatible type to previous declaration

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The specified identifier was already declared with different type modifiers. If the option [-Ansi](#) is enabled, this warning becomes an error.

Example

```
int i;  
int i();
```

Tips

Use the same type modifiers for the first declaration and the redeclaration.

C1021: Bit field type is not 'int'

[ERROR]

Description

Another type than ‘int’ was used for the bitfield declaration. Some Back Ends may support non-int bitfields, but only if the Compiler switch [-Ansi](#) is not given.

Example

```
struct {  
    char b:1;  
} S;
```

Tips

Use ‘int’ type for bitfields or remove the [-Ansi](#) from the Compiler options.

See also

Message C1106

C1022: 'far' used in illegal context

[ERROR]

Description

'far', 'rom' or 'uni' has been specified for an array parameter where it is not legal to use it. In ANSI C, passing an array to a function always means passing a pointer to the array, because it is not possible to pass an array by value. To indicate that the pointer is a non-standard pointer, non-standard keywords as 'near' or 'far' may be specified if supported.

Example

```
void foo(int far a) {} /* error */  
void foo(ARRAY far ap) {} /* ok: passing a far pointer */
```

Tips

Remove the illegal modifier.

C1023: 'near' used in illegal context

[ERROR]

Description

'far', 'rom' or 'uni' has been specified for an array parameter where it is not legal to use it. In ANSI C, passing an array to a function always means passing a pointer to the array, because it is not possible to pass an array by value. To indicate that the pointer is a non-standard pointer, non-standard keywords as 'near' or 'far' may be specified if supported.

Example

```
void foo(int near a) {} /* error */  
void foo(ARRAY near ap) {} /* ok: passing a near pointer */
```

Tips

Remove the illegal modifier.

C1024: Illegal bit field width

[ERROR]

Description

The type of the bit field is too small for the number of bits specified.

Example

```
struct {  
    int b:1234;  
} S;
```

Tips

Choose a smaller number of bits, or choose a larger type of the bitfield (if the backend allows such a non-standard extension).

C1025: ',' expected before '...'

[ERROR]

Description

An open parameter list was declared without a ',' before the '...'.

Example

```
void foo(int a ...);
```

Tips

Insert a ‘,’ before the ‘...’.

C1026: Constant must be initialized

[ERROR], C++

Description

The specified identifier was declared as const but was not initialized.

Example

```
const int i;           // error
extern const int i;   // ok
```

Tips

Initialize the constant object, or remove the ‘const’ specifier from the declaration.

C1027: Reference must be initialized

[ERROR], C++

Description

A reference was not initialized when it was declared.

Example

```
int j;
int& i /* = j; missing */
```

Tips

Initialize the reference with an object of the same type as the reference points to.

C1028: Member functions cannot be initialized

[ERROR] , C++

Description

A member function of the specified class was initialized.

Tips

Do not initialize member functions in the initialization list for a class or structure.

C1029: Undefined class

[ERROR] , C++

Description

A class is used which is not defined/declared.

Example

```
class A;  
class B {  
    A::I i; // error  
};
```

Tips

Define/declare a class before using it.

C1030: Pointer to reference illegal

[ERROR] , C++

Description

A pointer to a reference was declared.

Example

```
void f(int & * p);
```

Tips

The variable must be dereferenced before a pointer to it can be declared.

C1031: Reference to reference illegal

[ERROR], C++

Description

A reference to a reference was declared.

Tips

This error can be avoided by using pointer syntax and declaring a reference to a pointer.

C1032: Invalid argument expression

[ERROR], C++

Description

The argument expression of a function call in a Ctor-Init list was illegal.

Example

```
struct A {  
    A(int i);  
};  
  
struct B : A {  
    B();  
};  
  
B::B() : A((3) { // error  
}
```

Tips

In the argument expression of a Ctor-Init function call, there must be the same number of "(" as ")".

C1033: Ident should be base class or data member

[ERROR], C++

Description

An object in an initialization list was not a base class or member.

Example

```
class A {  
    int i;  
    A(int j) : B(j) {} // error  
};
```

Tips

Only a member or base class can be in the initialization list for a class or structure.

C1034: Unknown kind of linkage

[ERROR], C++

Description

The indicated linkage specifier was not legal. This error is caused by using a linkage specifier that is not supported.

Example

```
extern "MODULA-2" void foo(void);
```

Tips

Only the "C" linkage specifier is supported.

C1035: Friend must be declared in class declaration

[ERROR] , C++

Description

The specified function was declared with the friend specifier outside of a class, structure or union.

Example

```
friend void foo(void);
```

Tips

Do not use the friend specifier outside of class, structure or union declarations.

C1036: Static member functions cannot be virtual

[ERROR] , C++

Description

A static member function was declared as virtual.

Example

```
class A {  
    static virtual f(void); // error  
};
```

Tips

Do not declare a static member function as virtual.

C1037: Illegal initialization for extern variable in block scope

[ERROR]

Description

A variable with extern storage class cannot be initialized in a function.

Example

```
void f(void) {  
    extern int i= 1;  
}
```

Tips

Initialize the variable, where it is defined.

C1038: Cannot be friend of myself

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

The friend specifier was applied to a function/class inside the scope resolution of the same class.

Example

```
class A {  
    friend A::f(); //treated by the  
}; // compiler as "friend f();"
```

Tips

Do not write a scope resolution to the same class for a friend of a class.

C1039: Typedef-name or ClassName expected

[ERROR]

Description

In the current context either a typedef name or a class name was expected.

Example

```
struct A {  
    int i;  
};  
void main() {  
    A *a;  
    a=new a; // error  
}
```

Tips

Write the ident of a type or class/struct tag.

C1040: No valid :: classname specified

[ERROR] , C++

Description

The specified identifier after a scope resolution was not a class, structure, or union.

Example

```
class B {  
    class A {  
        };  
    };  
    class C : B::AA { // AA is not valid  
};
```

Tips

Use an identifier of an already declared class, structure, or union.

C1041: Multiple access specifiers illegal

[ERROR], C++

Description

The specified base class had more than one access modifier.

Tips

Use only one access modifier (public, private or protected).

C1042: Multiple virtual declaration makes no sense

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

The specified class or structure was declared as virtual more than once.

Tips

Use only one virtual modifier for each base class.

C1043: Base class already declared in base list

[ERROR], C++

Description

The specified class (or structure) appeared more than once in a list of base classes for a derived class.

Example

```
class A {};
class B: A, A {  
};
```

Tips

Specify a direct base class only once.

C1044: User defined Constructor is required

[ERROR], C++

Description

A user-defined constructor should be defined. This error occurs when a constructor should exist, but cannot be generated by the Compiler for the class.

Example

```
class A {  
    const int i;  
};
```

The compiler can not generate a constructor because he does not know the value for i.

Tips

Define a constructor for the class.

C1045: <Special member function> not generated

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

The Compiler option [-Cn=Ctr](#) disabled the creation of Compiler generated special member functions (Copy Constructor, Default Constructor, Destructor, Assignment operator).

Tips

If you want the special member functions to be generated by the Compiler, disable the Compiler option [-Cn=Ctr](#).

C1046: Cannot create compiler generated <Special member function> for nameless class

[ERROR], C++

Description

The Compiler could not generate a special member function (Copy Constructor, Default Constructor, Destructor, Assignment operator) for the nameless class.

Example

```
class B {  
public:  
    B();  
};  
class {  
    B b;  
} A;
```

Tips

Give a name to nameless class.

C1047: Local compiler generated <Special member function> not supported

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

Local class declarations would force the compiler to generate local special member functions (Copy Constructor, Default Constructor, Destructor, Assignment operator). But local functions are not supported.

Example

;

Tips

If you really want the compiler generated special member functions to be created, then declare the class (or struct) in the global scope.

C1048: Generate compiler defined <Special member function>

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

A special member function (Copy Constructor, Default Constructor, Destructor, Assignment operator) was created by the compiler.

When a class member or a base class contains a Constructor or a Destructor, then the new class must also have this special function so that the base class Constructor or Destructor is called in every case. If the user does not define one, then the compiler automatically generates one.

Example

```
struct A {  
    A(void);  
    A(const A&);  
    A& operator =(const A& );
```

```
    ~A( );
}
struct B : A {
};
```

Tips

If you do not want the compiler to generate the special member functions, then enable the option [-Cn=Ctr](#).

The compiler only calls a compiler generated function if it is really necessary. Often a compiler generated function is created, but then never called. Then the smart linker does not waste code space for such functions.

C1049: Members cannot be extern

[ERROR] , C++

Description

Class member cannot have the storage class "extern".

Example

```
class A {
    extern int f();
};
```

Tips

Remove the "extern" specifier.

C1050: Friend must be a class or a function

[ERROR] , C++

Description

The "friend"-specifier can only be used for classes (or structures or unions) and functions.

Example

```
typedef int I;  
struct A {  
    friend I; // illegal  
};
```

Tips

Use the "friend" specifier only for classes (or structures or unions) and functions.

C1051: Invalid function body

[ERROR]

Description

The function body of a member function inside a class declaration is invalid.

Example

```
struct A {  
    void f() { {{int i; }  
};
```

Tips

The function body of a member function inside a class declaration must have the same number of "{" as "}".

C1052: Unions cannot have class/struct object members containing Con/Destructor/Assign-Operator

[ERROR], C++

Description

The specified union member was declared with a special member (Con/Destructor/Assign-Operator).

Example

```
class A {  
    A(void);  
};  
  
union B {  
    A a;  
};
```

Tips

The union member may contain only compiler generated special members. So try to compile with the option [-Cn=Ctr](#) enabled.

C1053: Nameless class cannot have member functions

[ERROR] , C++

Description

A function was declared in a nameless class.

Example

```
class {  
    void f(void);  
};
```

Tips

Name the nameless class, or remove all member functions of the nameless class.

C1054: Incomplete type or function in class/struct/union

[ERROR] , C++

Description

A used type in a function, class, struct or union was not completely defined.

Example

Tips

Define types before using them.

C1055: External linkage for class members not possible

[ERROR], C++

Description

Member redeclarations cannot have external linkage.

Example

```
struct A {  
    f();  
} a;  
  
extern "C" A::f() {return 3;}
```

Tips

Do not declare members as extern.

C1056: Friend specifier is illegal for data declarations

[ERROR], C++

Description

The friend specifier cannot be used for data declarations.

Example

```
class A {  
    friend int a;  
};
```

Tips

Remove the friend specifier from the data declaration.

C1057: Wrong return type for <FunctionKind>

[ERROR] , C++

Description

The declaration of a function of *FunctionKind* contained an illegal return type.

Tips

Depending on FunctionKind:

- operator \rightarrow must return a pointer or a reference or an instance of a class, structure or union
- operator delete must return "void"
- operator new must return "void *"

C1058: Return type for FunctionKind must be <ReturnType>

[ERROR] , C++

Description

Some special functions must have certain return types. An occurred function did have an illegal return type.

Tips

Depending on FunctionKind:

- operator \rightarrow must return a pointer or a reference or an instance of a class, structure or union
- operator delete must return "void"
- operator new must return "void *"

C1059: Parameter type for <FunctionKind> parameter <No> must be <Type>

[ERROR], C++

Description

The declaration of a function of FunctionKind has a parameter of a wrong type.

Tips

Depending on FunctionKind:

- operator new parameter 1 must be “`unsigned int`”
- operator delete parameter 1 must be “`void *`”
- operator delete parameter 2 must be “`unsigned int`”

C1060: <FunctionKind> wrong number of parameters

[ERROR], C++

Description

The declaration of a function of FunctionKind has a wrong number of parameters

Tips

Depending on FunctionKind:

- member operator delete must have one or two parameters

C1061: Conversion operator must not have return type specified before operator keyword

[ERROR], C++

Description

A user-defined conversion cannot specify a return type.

Example

```
class A {  
public:  
    int operator int() {return value;} // error  
    operator int() {return value;} // ok  
private:  
    int value;  
}
```

Tips

Do not specify a return type before the operator keyword.

C1062: Delete can only be global, if parameter is (void *)

[ERROR], C++

Description

Global declarations/definitions of operator delete are allowed to have only one parameter.

Tips

Declare only one parameter for the global delete operator.

Or declare the delete operator as a class member, where it can have 2 parameters.

C1063: Global or static-member operators must have a class as first parameter

[ERROR], C++

Description

The specified overloaded operator was declared without a class parameter.

Example

```
int operator+ (int, int); // error;
```

Tips

The first parameter must be of class type.

C1064: Constructor must not have return type

[ERROR], C++

Description

The specified constructor returned a value, or the class name is used for a member.

Example

```
struct C {  
    int C(); // error  
    C(); // ok  
};
```

Tips

Do not declare a return type for constructors.

C1065: 'inline' is the only legal storage class for Constructors

[ERROR], C++

Description

The specified constructor has an illegal storage class (auto, register, static, extern, virtual).

Tips

The only possible storage class for constructors is "inline".

C1066: Destructor must not have return type

[ERROR], C++

Description

The specified destructor returned a value.

Example

Tips

Do not declare a return type for destructors.

C1067: Object is missing decl specifiers

[ERROR]

Description

An object was declared without decl-specifiers (type, modifiers, ...). There is no error, if compiling C-source without the [-ANSI](#) option.

Example

i

Tips

Apply decl-specifiers for the object, or compile without the options [-ANSI](#) and [-C++](#).

C1068: Illegal storage class for Destructor

[ERROR], C++

Description

The specified destructor has an illegal storage class (static, extern).

Tips

Do not use the storage classes "static" and "extern" for destructors.

C1069: Wrong use of far/near/rom/uni/paged in local scope

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

The far/near/rom/uni/paged keyword has no effect in the local declaration of the given identifier. 'far' may be used to indicate a special addressing mode for a global variable only. Note that such additional keywords are not ANSI compliant and not supported on all targets.

Example

```
far int i; // legal on some targets
void foo(void) {
    far int j; // error message C1069
}
```

Tips

Remove the far/near/rom/uni/paged qualifier, or declare the object in the global scope.

C1070: Object of incomplete type

[ERROR]

Description

An Object with an undefined or not completely defined type was used.

Example

```
void f(struct A a) {  
}
```

Tips

Check the spelling of the usage and the definition of this type. It is legal in C to pass a pointer to a undefined structure, so examine if is possible to pass a pointer to this type rather than the value itself.

C1071: Redefined extern to static

[ERROR]

Description

An extern identifier was redefined as static.

Example

```
extern int i;  
static int i; // error
```

Tips

Remove either the ‘extern’ specifier of the first declaration, or the ‘static’ specifier of the second occurrence of the identifier.

C1072: Redefined extern to static

[DISABLE, INFORMATION, WARNING, ERROR]

Description

If the option [-ANSI](#) is disabled, the nonstandard extension issues only a warning, not an error.

Example

```
extern int i;  
static int i; // warning
```

Tips

Remove either the ‘extern’ specifier of the first declaration, or the ‘static’ specifier of the second occurrence of the identifier.

C1073: Linkage specification contradicts earlier specification

[ERROR], C++

Description

The specified function was already declared with a different linkage specifier. This error can be caused by different linkage specifiers found in include files.

Example

```
int f(int i);  
extern "C" int f(int i);
```

Tips

Use the same linkage specification for the same function/variable.

C1074: Wrong member function definition

[ERROR] , C++

Description

The specified member function was not declared in the class/structure for the given parameters.

Example

```
class A {  
    void f(int i);  
};  
void A::f(int i, int i) { // error  
}
```

Tips

Check the parameter lists of the member function declarations in the class declaration and the member function declarations/definitions outside the class declaration.

C1075: Typedef object id already used as tag

[ERROR] , C++

Description

The identifier was already used as tag. In C++, tags have the same namespace than objects. So there would be no name conflict compiling in C.

Example

```
typedef const struct A A; // error in C++, ANSI-C ok
```

Tips

Compile without the option [-C++](#), or choose another name for the typedef object id.

C1076: Illegal scope resolution in member declaration

[ERROR], C++

Description

An access declaration was made for the specified identifier, but it is not a member of a base class.

Example

```
struct A {  
    int i;  
};  
  
struct B {  
    int j;  
};  
  
struct C : A {  
    A::i; // ok  
    B::j; // error  
};
```

Tips

Put the owner class of the specified member into the base class list, or do without the access declaration.

C1077: <FunctionKind> must not have parameters

[ERROR], C++

Description

Parameters were declared where it is illegal.

Tips

Do not declare parameters for Destructors and Conversions.

C1078: <FunctionKind> must be a function

[ERROR] , C++

Description

A constructor, destructor, operator or conversion operator was declared as a variable.

Example

```
struct A {  
    int A;  
};
```

Tips

Constructors, destructors, operators and conversion operators must be declared as functions.

C1080: Constructor/destructor: Parenthesis missing

[ERROR] , C++

Description

A redeclaration of a constructor/destructor is done without parenthesis.

Example

```
struct A {  
    ~A();  
};  
  
A::~A; // error  
A::~A(); // ok
```

Tips

Add parenthesis to the redeclaration of the constructor/destructor.

C1081: Not a static member

[ERROR], C++

Description

The specified identifier was not a static member of a class or structure.

Example

```
struct A {  
    int i;  
};  
void main() {  
    A::i=4; // error  
}
```

Tips

Use a member access operator (. or ->) with a class or structure object; or declare the member as static.

C1082: <FunctionKind> must be non-static member of a class/struct

[ERROR], C++

Description

The specified overloaded operator was not a member of a class, structure or union, and/or was declared as static. FunctionKind can be a conversion or an operator =, -> or ().

Tips

Declare the function inside a class declaration without the "static" storage class.

C1084: Not a member

[ERROR] , C++

Description

An ident has been used which is not a member of a class or a struct field.

Tips

Check the struct/class declaration.

C1085: <ident> is not a member

[ERROR] , C++

Description

A nonmember of a structure or union was incorrectly used.

Example

```
struct A {  
    int i;  
};  
void main() {  
    A a;  
    a.r=3; // error  
}
```

Tips

Using . or ->, specify a declared member.

C1086: Global unary operator must have one parameter

[ERROR] , C++

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Global member unary operator must have exactly one parameter.

C1087: Static unary operator must have one parameter

[ERROR] , C++

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Static member unary operator must have exactly one parameter.

C1088: Unary operator must have no parameter

[ERROR] , C++

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Member unary operator must have no parameters.

C1089: Global binary operator must have two parameters

[ERROR] , C++

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Global binary operator must have two parameters.

C1090: Static binary operator must have two parameters

[ERROR] , C++

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Static binary operator must have two parameters.

C1091: Binary operator must have one parameter

[ERROR] , C++

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Binary operator must have one parameter.

C1092: Global unary/binary operator must have one or two parameters

[ERROR], C++

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Global unary/binary operator must have one or two parameters.

C1093: Static unary/binary operator must have one or two parameters

[ERROR], C++

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Static unary/binary operator must have one or two parameters.

C1094: Unary/binary operator must have no or one parameter

[ERROR], C++

Description

The specified overloaded operator was incorrectly declared with the wrong number of parameters.

Tips

Unary/binary operator must have no or one parameter.

C1095: Postfix `++/-` operator must have integer parameter

[ERROR] , C++

Description

The specified overloaded operator was incorrectly declared with the wrong type of parameters.

Tips

Postfix `++/-` operator must have integer parameter.

C1096: Illegal index value

[ERROR]

Description

The index value of an array declaration was equal or less than 0.

Example

```
int i[0]; // error;
// for 16bit int this is 0x8CA0 or -29536 !
static char dct_data[400*90];
```

Tips

The index value must be greater than 0. If the index value is a calculated one, use a ‘u’ to make the calculation unsigned (e.g. 400*90u).

C1097: Array bounds missing

[ERROR]

Description

The non-first dimension of an array has no subscript value.

Example

```
int i[3][]; // error  
int j[][4]; // ok
```

Tips

Specify a subscript value for the non-first dimensions of an array.

C1098: Modifiers for non-member or static member functions illegal

[ERROR], C++

Description

The specified non-member function was declared with a modifier.

Example

```
void f() const; // error;
```

Tips

Do not use modifiers on non-member or static member functions.

C1099: Not a parameter type

[ERROR]

Description

An illegal type specification was parsed.

Example

```
struct A {  
    int i;  
};  
void f(A::i); // error
```

Tips

Specify a correct type for the parameter.

C1100: Reference to void illegal

[ERROR], C++

Description

The specified identifier was declared as a reference to void.

Example

```
void &vr; // error;
```

Tips

Don not declare references to a void type.

C1101: Reference to bitfield illegal

[ERROR], C++

Description

A reference to the specified bit field was declared.

Example

```
struct A {  
    int &i : 3; // error  
};
```

Tips

Do not declare references to bitfields.

C1102: Array of reference illegal

[ERROR], C++

Description

A reference the specified array was declared.

Example

```
extern int &j[20];
```

Tips

Do not declare references to arrays.

C1103: Second C linkage of overloaded function not allowed

[ERROR], C++

Description

More than one overloaded function was declared with C linkage.

Example

```
extern "C" void f(int);
extern "C" void f(long);
```

Tips

When using C linkage, only one form of a given function can be made external.

C1104: Bit field type is neither integral nor enum type

[ERROR]

Description

Bit fields must have an integral type (or enum type for C).

Example

```
struct A {  
    double d:1;  
};
```

Tips

Specify an integral type (int, long, short, ...) instead of the non-integral type.

C1105: Backend does not support non-int bitfields

[ERROR]

Description

Bit fields must be of integer type. Any other integral or non-integral type is illegal.

Some backends support non int bitfields, others do not. See the chapter backend for details.

Example

```
struct A {  
    char i:2;  
}
```

When the actual backend supports non-int bitfields, this message does not occur.

Tips

Specify an integer-type (int, signed int or unsigned int) for the bitfield.

C1106: Non-standard bitfield type

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Some of the Back Ends allow bitfield structure members of any integral type.

Example

```
struct bitfields {  
    unsigned short j:4; // warning  
};
```

Tips

If you want portable code, use an integer-type for bitfields.

C1107: Long long bit fields not supported yet

[ERROR]

Description

Long long bit fields are not yet supported.

Example

```
struct A {  
    long long l:64;  
};
```

Tips

Do not use long long bit fields.

C1108: Constructor cannot have own class/struct type as first and only

parameter

[ERROR], C++

Description

A constructor was declared with one parameter, and the parameter has the type of the class itself.

Example

```
struct B {  
    B(B b); // error  
};
```

Tips

Use a reference of the class instead of the class itself.

C1109: Generate call to Copy Constructor

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

An instance of a class was passed as argument of a function, needing to be copied onto the stack with the Copy Constructor.

Or an instance of a class was used as initializer of another instance of the same class, needing to be copied to the new class instance with the Copy Constructor

Example

```
struct A {  
    A(A &);  
    A();  
};  
  
void f(A a);  
  
void main(void) {  
    A a;  
    f(a); // generate call to copy ctor  
}
```

Tips

If conventional structure copying is desired, try to compile with the option [-Cn=Ctr](#) and do not declare copy constructors manually.

C1110: Inline specifier is illegal for data declarations

[ERROR], C++

Description

The "inline" specifier was used for a data declaration.

Example

```
inline int i;
```

Tips

Do not use "inline" specifiers for data declarations.

C1111: Bitfield cannot have indirection

[ERROR]

Description

A bitfield declaration must not contain a "*". There are no pointer to bits in C. Use instead a pointer to the structure containing the bitfield.

Example

```
struct bitfield {
    int *i : 2; // error
};
```

Tips

Do not use pointers in bitfield declarations.

C1112: Interrupt specifier is illegal for data declaration

[ERROR]

Description

The interrupt specifier was applied to a data declaration.

Example

```
interrupt int i; // error
```

Tips

Apply the interrupt specifier for functions only

C1113: Interrupt specifier used twice for same function

[ERROR]

Description

The interrupt specifier was used twice for the same function.

Example

```
interrupt 4 void interrupt 2 f(); // error
```

Tips

C1114: Illegal interrupt number

[ERROR]

Description

The specified vector entry number for the interrupt was illegal.

Example

```
interrupt 1000 void f(); // error
```

Tips

Check the backend for the range of legal interrupt numbers!

The interrupt number is not the same as the address of the interrupt vector table entry.

The mapping from the interrupt number to the vector address is backend specific.

C1115: Template declaration must be class or function

[ERROR], C++

Description

A non-class/non-function was specified in a template declaration.

Example

```
template<class A> int i; // error
```

Tips

Template declarations must be class or function

C1116: Template class needs a tag

[ERROR], C++

Description

A template class was specified without a tag.

Example

```
template<class A> struct { // error
    A a;
};
```

Tips

Use tags for template classes.

C1117: Illegal template/non-template redeclaration

[ERROR], C++

Description

An illegal template/non-template redeclaration was found.

Example

```
template<class A> struct B {
    A a;
};

struct B { // error
    A a;
};
```

Tips

Correct the source. Protect header files with from multiple inclusion.

C1118: Only bases and class member functions can be virtual

[ERROR], C++

Description

Because virtual functions are called only for objects of class types, you cannot declare global functions or union member functions as 'virtual'.

Example

```
virtual void f(void); // ERROR: definition of a global
                      // virtual function.
union U {
    virtual void f(void); // ERROR: virtual union member
                          // function
};
```

Tips

Do not declare a global function or a union member function as virtual.

C1119: Pure virtual function qualifier should be (=0)

[ERROR], C++

Description

The '=0' qualifier is used to declare a pure virtual function. Following example shows an ill-formed declaration.

Example

```
class A{      // class
public:
    virtual void f(void)=2; // ill-formed pure virtual
                           // function declaration.
};
```

Tips

Correct the source.

C1120: Only virtual functions can be pure

[ERROR], C++

Description

Only a virtual function can be declared as pure. Following example shows an ill-formed declaration.

Example

```
class A{           // class
public:
    void f(void)=0; // ill-formed declaration.
};
```

Tips

Make the function virtual. For overloaded functions check if the parameters are identical to the base virtual function.

C1121: Definition needed if called with explicit scope resolution

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

A definition for a pure virtual function is not needed unless explicitly called with the qualified-id syntax (nested-name-specifier [template] unqualified-id).

Example

```
class A{
public:
    virtual void f(void) = 0; // pure virtual function.
};

class B : public A{
public:
    void f(void){ int local=0; }
};

void main(void){
    B b;
    b.A::f();           // generate a linking error
    cause
        // no object is defined.
```

```
b.f(); // call the function defined in  
       // B class.  
}
```

Tips

Correct the source.

C1122: Cannot instantiate abstract class object

[ERROR], C++

Description

An abstract class is a class that can be used only as a base class of some other class; no objects of an abstract class may be created except as objects representing a base class of a class derived from it.

Example

```
class A{  
public:  
    virtual void f(void) = 0; //pure virtual function ==> A  
                           //is an abstract class  
};  
void main(void){  
    A a;  
}
```

Tips

- Use a pointer/reference to the object:

```
void main(void){  
    A *pa;  
}
```

- Use a derived class from abstract class:

```
class B : public A{  
public:
```

```
    void f(void){}
}
void main(void){
    B b;
}
```

C1123: Cannot instantiate abstract class as argument type

[ERROR], C++

Description

An abstract class may not be used as argument type

Example

```
class A{
public:
    virtual void f(void) = 0; // pure virtual function
                           // ==> A is an abstract class
}
void main(void){
    void fct(A);
}
```

Tips

- Use a pointer/reference to the object:

```
void main(void){
    void fct(A *);
}
```

- Use a derived class from abstract class

```
class B : public A{
public:
    void f(void){}
}

void main(void){
    void fct(B);
}
```

C1124: Cannot instantiate abstract class as return type

[ERROR], C++

Description

An abstract class may not be used as a function return type.

Example

```
class A{
public:
    virtual void f(void) = 0; //pure virtual function ==> A
                           //is an abstract class
};

void main(void){
    A fct(void);
}
```

Tips

- Use a pointer/reference to the object

```
void main(void){
    A *fct(void);
}
```

- Use a derived class from abstract class

```
class B : public A{
public:
    void f(void){}
};

void main(void){
    B fct(void);
}
```

C1125: Cannot instantiate abstract class as a type of explicit conversion

[ERROR], C++

Description

An abstract class may not be used as a type of an explicit conversion.

Example

```
class A{
public:
    virtual void f(void) = 0; //pure virtual function ==> A
is an abstract class
};

class B : public A{
public:
    void f(void){}
};

void main(void){
    A *pa;
    B b;

    pa = &(A)b;
}
```

Tips

Use a pointer/reference to the object

```
void main(void){
    A *pa;
    B b;

    pa = (A *)b;
}
```

C1126: Abstract class cause inheriting pure virtual without overriding function(s)

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

Pure virtual functions are inherited as pure virtual functions.

Example

```
class A{
public:
    virtual void f(void) = 0;
    virtual void g(void) = 0;
};

class B : public A{
public:
    void f(void){}
    // void B::g(void) is inherited pure virtual
    // ==> B is an implicit abstract class
};
```

C1127: Constant void type probably makes no sense

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

A pointer/reference to a constant void type was declared

Example

```
const void *cvp;    // warning (pointer to constant void)
void *const vpc;   // no warning (constant pointer)
```

Tips

A pointer to a constant type is a different thing than a constant pointer.

C1128: Class contains private members only

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

A class was declared with only private members.

Example

```
class A {  
    private:  
        int i;  
};
```

Tips

You can never access any member of a class containing only private members!

C1129: Parameter list missing in pointer to member function type

[ERROR], C++

Description

The current declaration is neither the one of pointer to member nor the one of pointer to member function. But something looking like melting of both.

Example

```
class A{  
public:  
    int a;  
};  
  
typedef int (A::*pm);
```

Tips

Use the standard declaration: for a pointer to member 'type class::*ident' and for a pointer to member function 'type (class::*ident)(param list)'

```
class A{  
public:  
    int a;  
    void fct(void);  
};  
  
typedef int A::*pmi;  
  
typedef void (A::*pmf)(void);
```

C1130: This C++ feature is disabled in your current cC++/EC++ configuration

[ERROR], C++

Description

Either the Embedded C++ language (EC++) is enabled ([option -C++e](#)), or the compactC++ language (cC++) is enabled ([option -C++c](#)) plus the appropriate feature is disabled ([option -Cn](#)).

Following features could be disabled:

- virtual functions
- templates
- pointer to member
- multiple inheritance and virtual base classes
- class parameters and class returns

Tips

If you really don't want to use this C++ feature, you have to find a workaround to the problem. Otherwise change the C++ language configuration with the options [-C++](#) and [-Cn](#), or use the advanced option dialog

C1131: Illegal use of global variable address modifier

[ERROR]

Description

The global variable address modifier was used for another object than a global variable.

Example

```
int glob @0xf90b; // ok, the global variable "glob" is
at 0xf90b

int *globp @0xf80b = &glob; // ok, the global variable
// "globp" is at 0xf80b and
// points to "glob"

void g() @0x40c0; // error (the object is a function)

void f() {
    int i @0x40cc; // error (the object is a local
variable)
}
```

Tips

Global variable address modifiers can only be used for global variables. They cannot be used for functions or local variables. Global variable address modifiers are not ANSI standard. So the option [-Ansi](#) has to be disabled.

To Put a function at a fixed address, use a pragma CODE_SEG to specify a segment for a function. Then map the function in the prm file with the linker.

To call a function at a absolute address use a cast:

```
#define f ((void (*) (void)) 0x100)

void main(void) {
    f();
}
```

C1132: Cannot define an anonymous type inside parentheses

[ERROR] , C++

Description

An anonymous type was defined inside parentheses. This is illegal in C++.

Example

```
void f(enum {BB,AA} a) { // C ok, C++ error
}
```

Tips

Define the type in the global scope.

C1133: Such an initialization requires STATIC CONST INTEGRAL member

[FATAL, ERROR, WARNING, INFORMATION], C++

Description

A static CONST member of integral type may be initialized by a constant expression in its member declaration within the class declaration.

Example

```
int e = 0;  
  
class A{  
public:  
    static int a = 1; // ERROR: non-const initialized  
    const int b = 2; // ERROR: non-static initialized  
    static const float c = 3.0;// ERROR: non-integral  
                           // initializer  
    static const int d = e; // ERROR: non-const initializer  
                           // ...  
};
```

Tips

```
class A{  
public:  
    static const int a = 5; // Initialization  
                           // ...  
};  
  
const int A::a;           // Definition
```

or the other way round:

```
class A{  
public:  
    static const int a;      // Definition
```

```
// ...
};

const int A::a = 5;           // Initialization
```

C1134: Static data members are not allowed in local classes

[ERROR], C++

Description

Static members of a local class have no linkage and cannot be defined outside the class declaration. It follows that a local class cannot have static data members.

Example

```
void foo(void){
    class A{
        public:
            static int a; // ERROR because static data member
            static void myFct(void); // OK because static method
    };
}
```

Tips

Remove the static specifier from the data member declarations of any local class.

```
void foo(void){
    class A{
        public:
            int a; // OK because data member.
            static void myFct(void); // OK because static method
    };
}
```

C1135: Ignore Storage Class Specifier cause it only applies on objects

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The specified Storage Class Specifier is not taken in account by the compiler, because it does not apply to an object.

Example

```
static class A{  
    public:  
        int a;  
};
```

Tips

Remove the Storage Class Specifier from the class definition and apply it to the instances.

```
class A{  
    public:  
        int a;  
};  
  
static A myClassA;
```

C1136: Class <Ident> is not a correct nested class of class <Ident>

[ERROR], C++

Description

Error detected while parsing the scope resolution of a nested class.

Example

```
class A{  
    class B;  
}  
  
class A::C{ };
```

Tips

Check that the scope resolution matches the nested class.

```
class A{  
    class B;  
}  
  
class A::B{ };
```

C1137: Unknown or illegal segment name

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A segment name used in a segment specifier was not defined with a segment pragma before.

Example

```
#pragma DATA_SEG AA  
  
#pragma DATA_SEG DEFAULT  
  
int i @ "AA"; // OK  
  
int i @ "BB"; // Error C1137, segment name BB not known
```

Tips

All segment names must be known before they are used to avoid tipping mistakes and to specify a place to put segment modifiers.

See also

[#pragma DATA_SEG](#)
[#pragma CONST_SEG](#)
[#pragma CODE_SEG](#)

C1138: Illegal segment type

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A segment name with an illegal type was specified. Functions can only be placed into segments of type CODE_SEG, variable/constants can only be placed into segments of type DATA_SEG or CONST_SEG.

Example

```
#pragma DATA_SEG AA
#pragma CODE_SEG BB
int i @ "AA"; // OK
int i @ "BB"; // Error C1138,
                // data cannot be placed in codeseg
```

Tips

Use different segment names for data and code. To place constants into rom, use segments of type CONST_SEG.

See also

[#pragma DATA_SEG](#)
[#pragma CONST_SEG](#)
[#pragma CODE_SEG](#)

C1139: Interrupt routine should not have any return value nor any parameter

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Interrupt routines should not have any return value nor any parameter. In C++, member functions cannot be interrupt routines due to hidden THIS parameter. Another problem may be that a [#pragma TRAP_PROC](#) is active where it should not be.

Example

```
int interrupt myFctl(void){  
    return 4;  
}  
  
#pragma TRAP_PROC  
  
void myFct2(int param){ }  
  
class A{  
public:  
    void myFctMbr(void);  
};  
  
void interrupt A::myFctMbr(void){}
```

Tips

Remove all return value and all parameter (even hidden, e.g. ‘this’ pointer for C++):

```
void interrupt myFctl(void){ }  
  
#pragma TRAP_PROC  
  
void myFct2(void){ }  
  
class A{  
public:  
    void myFctMbr(void);  
};  
  
void A::myFctMbr(void){ }  
  
void interrupt myInterFct(void){ }
```

C1140: This function is already declared and has a different prototype

[DISABLE, INFORMATION, WARNING, ERROR]

Description

There are several different prototypes for the same function in a C module.

Example

```
int Foo (char,float,int,int* );
int Foo (char,float,int,int**);
int Foo (char,char,int,int**);
```

Tips

Check which one is correct and remove the other(s).

C1141: Ident <ident> cannot be allocated in global register

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The global variable <ident> cannot be allocated in the specified register. There are two possible reasons:

- The type of the variable is not supported to be accessed in a register.
or
- The specified register number is not possible (e.g. used for parameter passing).

Example

```
extern int glob_var @_REG 4; //r4 is used for parameters
```

Tips

Consider the ABI of the target processor.

C1142: Invalid Cosmic modifier. Accepted: @near, @far, @tiny or @interrupt (-ANSI off)

[ERROR]

Description

The modified after the @ was not recognized.

Example

```
@nearer unsigned char index;
```

Tips

Check the spelling.

Cosmic modifiers are only supported with the option -Ccx.

Not all backends do support all qualifiers.

Consider using a #pragma DATA_SEG with a qualifier instead.

C1143: Ambiguous Cosmic space modifier. Only one per declaration allowed

[[ERROR](#)]

Description

Multiple cosmic modifiers were found for a single declaration. Use only one of them.

Example

```
@near @far unsigned char index;
```

Tips

Cosmic modifiers are only supported with the option -Ccx.

Not all backends do support all qualifiers.

Only use one qualifier.

Consider using a #pragma DATA_SEG with a qualifier instead.

C1144: Multiple restrict declaration makes no sense

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The restrict qualifier should only be applied once and not several times.

Example

```
int * restrict restrict pointer;
```

Tips

Only specify restrict once.

C1390: Implicit virtual function

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

The 'virtual' keyword is needed only in the base class's declaration of the function; any subsequent declarations in derived classes are virtual by default.

Example

Example 1:

```
class A{ //base class
public:
    virtual void f(void){ glob=2; } //definition of a
                                    //virtual function
};

class B : public A{ //derived class
public:
    void f(void){ glob=3; } //overriding function
                            //IMPLICIT VIRTUAL FUNCTION
};
```

Example 2:

```
class A{ // base class
public:
    virtual void f(void){ glob=2; } //definition of a
                                    //virtual function.
};

class B : public A{ //derived class
public:
    virtual void f(void){ glob=3; } //overriding function:
                                    //'virtual' is not
                                    // necessary here
};
```

Example 3:

```
class A{ //base class
public:
    virtual void f(void){ glob=2; } //definition of a
                                    //virtual function.
};

class B : public A { //derived class
public:
    void f(int a){ glob=3+a; } // not overriding function
};
```

Tips

A derived class's version of a virtual function must have the same parameter list and return type as those of the base class. If these are different, the function is not considered a redefinition of the virtual function. A redefined virtual function cannot differ from the original only by the return type.

C1391: Pseudo Base Class is added to this class

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

The 'virtual' keyword ensures that only one copy of the subobject is included in the memory scope of the object. This single copy is the PSEUDO BASE CLASS.

Example

```
class A{ //base class
    // member list
};

class B : public virtual A { //derived class
public:
    // member list
};

class C : public virtual A { //derived class
public:
    // member list
};

class D : public B, public C { //derived class
public:
    // member list
};
```

Tips

According to this definition, an object 'd' would have the following memory layout:

```
A part
B part
-----
A part
C part
-----
D part
```

But the 'virtual' keyword makes the compiler to generate the following memory layout.

```
B part
-----
C part
-----
A part      // This is the PSEUDO BASE CLASS of class D
```

D part

In addition, a pointer to the PSEUDO BASE CLASS is included in each base class that previously derived from virtual base class (here B and C classes).

C1392: Pointer to virtual methods table not qualified for code address space (use [-Qvtrom](#) or [-Qvtpuni](#))

[ERROR] , C++

Description

If a virtual methods table of a class is forced to be allocated in code address space (only possible with Harvard architecture), the pointers to the virtual methods table must be qualified with a 'rom' attribute (i.e. rom pointer). This message currently appears only if you specify the compiler option [-Cc](#) (allocate 'const' objects in rom). For Harvard targets all virtual methods tables are put into code address space because the virtual methods tables are always constant. In this case the compiler generated pointers to the virtual methods table must be qualified with the 'rom'- or 'uni' attribute (see under Tips).

Tips

Qualify virtual table pointers with 'rom' by setting the compiler option [-Qvtrom](#).

See also

Option [-Qvtp](#)

C1393: Delta value does not fit into range (option [-Tvtd](#))

[ERROR] , C++

Description

An option '-Tvtd' is provided by the compiler to let the user specify the delta value size. The delta value is used in virtual functions management in order to set up the value of the THIS pointer. Delta value is stored in virtual tables.

Letting the user specify the size of the delta value can save memory space. But one can imagine that the specified size can be too short, that is the aim of this error message.

Example

```
class A{
public:
    long a[33]
};

class B{
public:
    void virtual fct2 (void){}
};

class C : public A, public B{
public:
};

void main (void){
    C c;
    c.fct2();
}
```

If the previous example is compiled using the option '-Tvtd1' then the minimal value allowed for delta is (-128) and a real value of delta is -(4*33)=-132.

Tips

Specify a larger size for the delta value: '-Tvtd2'.

See also option [-T, C++ Front End](#)

C1395: Classes should be the same or derive one from another

[ERROR], C++

Description

Pointer to member is defined to point on a class member, then classes (the one which member belongs to and the one where the pointer to member points) have to be identical. If the member is inherited from a base class then classes can be different.

Example

```
class A{
public:
    int a;
    void fct1(void){}
};

class B{
public:
    int b;
    void fct2(void){}
};

void main(void){
    int B::*pmi = &A::a;
    void (B::*pmf)() = &A::fct1;
}
```

Tips

Use the same classes

```
class A{
public:
    int a;
    void fct1(void){}
};

class B{
public:
    int b;
    void fct2(void){}
};

void main(void){
    int A::*pmi = &A::a;
    void (A::*pmf)() = &A::fct1;
}
```

Use classes which derive one from another

```
class A{
public:
    int a;
```

```
        void fct1(void){ }
    };
    class B : public A{
public:
    int b;
    void fct2(void){ }
};
void main(void){
    int B::*pmi = &A::a;
    void (B::*pmf)() = &A::fct1;
}
```

C1396: No pointer to STATIC member: use classic pointer

[ERROR], C++

Description

Syntax of pointer to member cannot be used to point to a static member. Static member have to be pointed in the classic way.

Example

```
int glob;

class A{
public:
    static int a;
    static void fct(void){}
};

void main(void){
    int A::*pmi = &A::a;
    void (A::*pmf)() = &A::fct;
}
```

Tips

Use the classic pointer to point static members

```
class A{
public:
    static int a;
```

```
        static void fct(void){ }
    };
    void main(void){
        A aClass;
        int *pmi = &aClass.a;
        void (*pmf)() = &aClass.fct;
    }
```

C1397: Kind of member and kind of pointer to member are not compatible

[ERROR], C++

Description

A pointer to member can not point to a function member and a pointer to function member can not point a member.

Example

```
class A{
public:
    int b;
    int c;
    void fct(){}
    void fct2(){}
};

void main(void){
    int A::*pmi = &A::b;
    void (A::* pmf)() = &A::fct;
    pmi=&A::fct2;
    pmf=&A::c;
}
```

Tips

```
class A{
public:
    int b;
    int c;
    void fct(){}
}
```

```
        void fct2(){}  
};  
void main(void){  
    int A::*pmi = &A::b;  
    void (A::* pmf)() = &A::fct;  
    pmf=&A::fct2;  
    pmi=&A::c;  
}
```

C1398: Pointer to member offset does not fit into range of given type (option -Tpmo)

[ERROR], C++

Description

An option -Tpmo is provided by the compiler to let the user specify the pointer to member offset value size. Letting the user specify the size of the offset value can save memory space. But one can imagine that the specified size is too short, that is the aim of this message.

Example

```
class A{  
public:  
    long a[33];  
    int b;  
};  
void main (void){  
    A myA;  
    int A::*pmi;  
  
    pmi = &A::b;  
    myA.*pmi = 5;  
}
```

If the previous example is compiled using the option -Tpmo1 then the maximal value allowed for offset is (127) and a real value of offset is (4*33)=132.

Tips

Specify a larger size for the offset value: -Tpmo2. See also option [-T, C++ Front End](#).

C1400: Missing parameter name in function head

[DISABLE, INFORMATION, WARNING, ERROR]

Description

There was no identifier for a name of the parameter. Only the type was specified. In function declarations, this is legal. But in function definitions, it's illegal.

Example

```
void f(int) {} // error
void f(int);    // ok
```

Tips

Declare a name for the parameter.

In C++ parameter names must not be specified.

C1401: This C++ feature has not been implemented yet

[ERROR], C++

Description

The C++ compiler does not support all C++ features yet. See also the C++ section about features that are not implemented yet.

Tips

Try to find a workaround of the problem or ask about the latest version of the compiler.

C1402: This C++ feature (<Feature>) is not implemented yet

[ERROR] , C++

Description

The C++ compiler does not support all C++ features yet.

Here is the list of features causing the error:

- Compiler generated functions of nested nameless classes
- calling destructors with goto
- explicit operator call
- Create Default Ctor of unnamed class
- Base class member access modification
- local static class obj
- global init with non-const

Tips

Try to avoid to use such a feature, e.g. using global static objects instead local ones.

C1403: Out of memory

[FATAL]

Description

The compiler wanted to allocate memory on the heap, but there was no space left.

Tips

Modify the memory management on your system.

C1404: Return expected

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A function with a return type has no return statement. In C it's a warning, in C++ an error.

Example

```
int f() {} // warning
```

Tips

Insert a return statement, if you want to return something, otherwise declare the function as returning 'void' type.

C1405: Goto <undeclared Label> in this function

[ERROR]

Description

A goto label was found, but the specified label did not exist in the same function.

Example

```
void main(void) {
    goto label;
}
```

Tips

Insert a label in the function with the same name as specified in the goto statement.

C1406: Illegal use of identifierList

[ERROR]

Description

A function prototype declaration had formal parameter names, but no types were provided for the parameters.

Example

```
int f(i); // error
```

Tips

Declare the types for the parameters.

C1407: Illegal function-redefinition

[ERROR]

Description

The function has already a function body, it has already been defined.

Example

```
void main(void) {}
void main(void) {}
```

Tips

Define a function only once.

C1408: Incorrect function-definition

[ERROR]

Description

The function definition is not correct or ill formed.

Tips

Correct the source.

C1409: Illegal combination of parameterlist and identlist

[ERROR]

Description

The parameter declaration for a function does not match with the declaration.

Tips

Correct the source. Maybe a wrong header file has been included.

C1410: Parameter-declaration - identifier-list mismatch

[ERROR]

Description

The parameter declaration for a function does not match with the declaration.

Tips

Correct the source. Maybe a wrong header file has been included.

C1411: Function-definition incompatible to previous declaration

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An old-style function parameter declaration was not compatible to a previous declaration.

Example

```
void f(int i);
void f(i,j) int i; int j; {} // error
```

Tip

Declare the same parameters as in the previous declaration.

C1412: Not a function call, address of a function

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A function call was probably desired, but the expression denotes an address of the function.

Example

```
void f(int i);
void main() {
    f; // warning
}
```

Tips

Write parenthesis "(,)" with the arguments after the name of the function to be called.

C1413: Illegal label-redeclaration

[ERROR]

Description

The label was defined more than once.

Example

```
Label:  
...  
Label: // label redefined
```

Tips

Choose another name for the label.

C1414: Casting to pointer of non base class

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

A cast was done from a pointer of a class/struct to a pointer to another class/struct, but the second class/struct is not a base of the first one.

Example

```
class A{} a;  
class B{};  
void main(void) {  
    B* b= (B*)&a;  
}
```

Tips

Check if your code really does what you want it to.

C1415: Type expected

[ERROR]

Description

The compiler cannot resolve the type specified or no type was specified. This message may also occur if no type is specified for a new operator.

Tips

Correct the source, add a type for the new operator.

C1416: No initializer can be specified for arrays

[ERROR]

Description

An initializer was given for the specified array created with the new operator.

Tips

Initialize the elements of the array after the statement containing the "new" operator.

C1417: Const/volatile not allowed for type of new operator

[ERROR]

Description

The new operator can only create non const and non volatile objects.

Example

```
void main() {
    int *a;
    typedef const int I;
    a=new I; // error
}
```

Tips

Do not use const/volatile qualifiers for the type given to the new operator.

C1418:] expected for array delete operator

[ERROR]

Description

There was no "]" found after the "[" of a delete operator.

Example

```
delete [] MyArray; // ok  
delete [3] MyArray; // error
```

Tips

Add a "]" after the "[".

C1419: Non-constant pointer expected for delete operator

[ERROR]

Description

A pointer to a constant object was illegally deleted using the delete operator.

Example

```
void main() {  
    const int *a;  
    a=new int;  
    delete a; // error  
}
```

Tips

The pointer to be deleted has to be non-constant.

C1420: Result of function-call is ignored

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A function call was done without saving the result.

Example

```
int f(void);  
void main(void) {  
    f(); /* ignore result */  
}
```

Tips

Assign the function call to a variable, if you need the result afterwards.

Otherwise cast the result to void. E.g.:

```
int f(void);  
void main(void) {  
    (void)f(); /* explicitly ignore result */  
}
```

C1421: Undefined class/struct/union

[ERROR]

Description

An undefined class, structure or union was used.

Example

```
void f(void) {  
    struct S *p, *p1;  
    *p=*p1;  
}
```

Tips

Define the class/struct/union.

C1422: No default Ctor available

[ERROR], C++

Description

No default constructor was available for the specified class/struct. The compiler will supply a default constructor only if user-defined constructors are not provided in the same class/struct and there are default constructors provided in all base/member classes/structs.

Example

```
class A {  
public:  
    A(int i); // constructor with non-void parameter  
    A();       // default constructor  
};
```

Tips

If you provide a constructor that takes a non-void parameter, then you must also provide a default constructor.

Otherwise, if you do not provide a default constructor, you must call the constructor with parameters.

Example

```
class A {  
public:  
    A(int i);  
};  
A a(3); // constructor call with parameters
```

C1423: Constant member must be in initializer list

[ERROR] , C++

Description

There are constant members in the class/struct, that are not initialized with an initializer list in the object constructor.

Example

```
struct A {  
    A();  
    const int i;  
};  
A::A() : i(4) { // initialize i with 4  
}
```

Tips

If a const or reference member variable is not given a value when it is initialized, it must be given a value in the object constructor.

C1424: Cannot specify explicit initializer for arrays

[ERROR] , C++

Description

The specified member of the class/struct could not be initialized, because it is an array.

Tips

Initialize the member inside the function body of the constructor.

C1425: No Destructor available to call

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

No destructor is available, but one must be called.

C1426: Explicit Destructor call not allowed here

[ERROR], C++

Description

Explicit Destructor calls inside member functions without using "this" are illegal.

Example

```
struct A {  
    void f();  
    ~A();  
};  
  
void A::f() {  
    ~A(); // illegal  
    this->~A(); // ok  
}
```

Tips

Use the this pointer.

C1427: 'this' allowed in member functions only

[ERROR], C++

Description

The specified global function did not have a this pointer to access.

Tips

Do not use "this" in global functions.

C1428: No wide characters supported

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler does not support wide characters. They are treated as conventional characters.

Example

```
char c= L'a'; // warning
```

Tips

Do not specify the "L" before the character/string constant.

C1429: Not a destructor id

[ERROR], C++

Description

Another name than the name of the class was used to declare a destructor.

Tips

Use the same name for the destructor as the class name.

C1430: No destructor in class/struct declaration

[ERROR], C++

Description

There was no destructor declared in the class/struct.

Example

```
struct A {  
};  
A::~A() {} // error  
void main() {  
    A.a;  
    a.~A(); // legal  
}
```

Tips

Declare a destructor in the class/struct.

C1431: Wrong destructor call

[ERROR], C++

Description

This call to the destructor would require the destructor to be static. But destructors are never static.

Example

```
class A {  
public:  
    ~A();  
    A();  
};  
void main() {  
    A::~A(); // error
```

```
A::A(); // ok, generating temporary object
}
```

Tips

Do not make calls to static destructors, because there are no static destructors.

C1432: No valid classname specified

[ERROR], C++

Description

The specified identifier was not a class/structure or union.

Example

```
int i;
void main() {
    i::f(); // error
}
```

Tips

Use a name of a class/struct/union.

C1433: Explicit Constructor call not allowed here

[ERROR], C++

Description

An explicit constructor call was done for a specific object.

Example

```
struct A {
    A();
```

```
    void f();
}
void A::f() {
    this->A(); // error
    A();        // ok, generating temporary object
}
void main() {
    A a;
    a.A();      // error
    A();        // ok, generating temporary object
}
```

Tips

Explicit constructor calls are only legal, if no object is specified, that means, a temporary object is generated.

C1434: This C++ feature is not yet implemented

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

The C++ compiler does not yet support all C++ features.

Here is a list of not yet implemented features causing a warning:

- Check for NULL ptr for this complex expression
- Class parameters (for some processors this is already implemented!)
- Class returns (for some processors this is already implemented!)

Tips

The C++ compiler ignores this C++ feature and behaves like a C-Compiler.

C1435: Return expected

[ERROR], C++

Description

The specified function was declared as returning a value, but the function definition did not contain a return statement.

Example

```
int foo(void) {}
```

Tips

Write a return statement in this function or declare the function with a void return type.

C1436: delete needs number of elements of array

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

A call to the `delete[]` operator was made without specifying the number of elements of the array, which is necessary for deleting a pointer to a class needing to call a destructor.

Example

```
class A {
    /* ... */
    ~A();
};

class B {
    /* ... */
};

void f(A *ap, B *bp) {
    delete ap;          // ok
    delete[] ap;        // error
    delete[4] ap;       // ok
    delete bp;          // ok
    delete[] bp;        // ok
    delete[4] bp;       // ok
}
```

Tips

Specify the number of elements at calling delete[].

C1437: Member address expected

[ERROR], C++

Description

A member address is expected to initialize the pointer to member. 0 value can also be provided to set the pointer to member to NULL.

Example

```
class A{
public:
    int a;
    void fct(void){ }
};

void main(void){
    int A::*pmi = NULL;
    ...
}
```

Tips

Use a member address

```
class A{
public:
    int a;
    void fct(void){ }
};

void main(void){
    int A::*pmi = &A::a;
    void (A::*pmf)() = &A::fct;
    ...
}
```

Use 0 value to set the pointer to member to NULL

```
class A{
public:
```

```
int a;
void fct(void){}
};

void main(void){
    int A::*pmi = 0;
    void (A::*pmf)() = 0;
    ...
}
```

C1438: ... is not a pointer to member ident

[ERROR], C++

Description

Parsing ident is not a pointer to member as expected.

Example

```
int glob;

class A{
public:
    int a;
    void fct(void){}
};

void main(void){
    int A::*pmi = &A::a;
    void (A::*pmf)() = &A::fct;
    A aClass;
    ...
    aClass.*glob = 4;
    (aClass.*glob)();
}
```

Tips

Use the pointer to member ident

```
class A{
public:
    int a;
```

```
    void fct(void){ }
}
void main(void){
    int A::*pmi = &A::a;
    void (A::*pmf)() = &A::fct;
    A aClass;
    ...
    aClass.*pmi = 4;
    (aclass.*pmf)();
}
```

C1439: Illegal pragma __OPTION_ACTIVE__, <Reason>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An ill formed __OPTION_ACTIVE__ expression was detected. The reason argument gives a more concrete hint what actually is wrong.

Example

```
#if __OPTION_ACTIVE__( "-dABS" )
#endif
```

The __OPTION_ACTIVE__ expression only allows the option to be tested (here “-d” and not the content of the option here “ABS”).

Tips

Only use the option. To check if a macro is defined as in the example above, use “#if defined(ABS)”. Only options known to the compiler can be tested. This option can be moved to an warning or less.

See also

__OPTION_ACTIVE__

C1440: This is causing previous message <MsgNumber>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message informs about the problem causing the previous message with the number <MsgNumber>. Because the reason for the previous message may be in another file (e.g. header file), this message helps to find out the problem.

Example

```
void foo(void);
int foo(void) {}
```

produces following messages:

```
int foo(void) {}
^ERROR C1019: Incompatible type to previous declaration
              (found 'int (*) ()', expected 'void (*) ()')
void foo(void);
^ INFORMATION C1440: This is causing previous message
1019
```

Tips

The problem location is either the one indicated by the previous message or the location indicated by this message.

C1441: Constant expression shall be integral constant expression

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A constant expression which has to be an integral expression is not integral. A non-integral expression is e.g. a floating constant expression.

Example

```
#if 1. /* << has to be integral! */
;
#endif
```

Tips

Use a integral constant expression. Note: if you move this message (to disable/information/warning), the non-integral constant expression is transformed into an integral expression (e.g. 2.3 => 2).

C1442: Typedef cannot be used for function definition

[ERROR]

Description

A typedef name was used for a function definition.

Example

```
typedef int INTFN();
INTFN f { return (0); } /* << error */
```

Tips

Do not use a typedef name for a function definition.

C1443: Illegal wide character

[ERROR]

Description

There is a illegal wide character after a wide character designator (“L”).

Example

```
int i = sizeof(L 3.5);
```

Tips

After “L” there has to be a character constant (e.g. L'a') or a string (e.g. L"abc").

C1444: Initialization of <Variable> is skipped by 'case' label

[DISABLE, INFORMATION, WARNING, ERROR] C++

Description

Initialization of a local variable is skipped by a 'case' label.

Example

```
void main(void){  
    int i;  
  
    switch(i){  
        int myVar = 5;  
        case 0:           // C1444 init skipped  
            //...  
            break;  
        case 1:           // C1444 init skipped  
            //...  
            break;  
    }  
}
```

Tips

Declare the local variable in the block where it is used.

```
void main(void){  
    int i;  
  
    switch(i){  
        case 0:  
    }
```

```
//...
break;
case 1:
    int myVar = 5;
    //...
    break;
}
```

C1445: Initialization of <Variable> is skipped by 'default' label

[DISABLE, INFORMATION, WARNING, ERROR] C++

Description

Initialization of a local variable is skipped by a 'default' label.

Example

```
void main(void){
    int i;

    switch(i){
        case 0:
            //...
            break;
        int myVar = 5;
        default:           // C1445 init skipped
            //...
            break;
    }
}
```

Tips

Declare the local variable in the block where it is used.

```
void main(void){
    int i;

    switch(i){
        case 0:
            //...
            break;
        default:
```

```
int myVar = 5;  
//...  
break;  
}
```

C1800: Implicit parameter-declaration (missing prototype) for '<FuncName>'

[ERROR]

Description

A function was called without its prototype being declared before.

Example

```
void f(void) {  
    g();  
}
```

This message is only used for C++ or for C if option [-Wpd](#), but not Option [-Ansi](#) is given

Tips

Prototype the function before calling it.

Use void to define a function with no parameters.

```
void f(); /* better: 'void f(void)' */  
void g(void);
```

The C declaration f does not define anything about the parameters of f. The first time f is used, the parameters get defined implicitly. The function g is defined to have no parameters.

C1801: Implicit parameter-declaration for '<FuncName>'

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A function was called without its prototype being declared before.

Example

```
void f(void) {  
    g();  
}
```

Tips

Prototype the function before calling it. Make sure that there is a prototype/declaration for the function. E.g. for above example:

```
void g(void); /* having correct prototype for 'g' */  
void f(void) {  
    g();  
}
```

C1802: Must be static member

[ERROR]

Description

A non-static member has been accessed inside a static member function.

Example

```
struct A {  
    static void f();  
    int i;  
};  
void A::f() {  
    i=3; // error  
}
```

Tips

Remove the "static" specifier from the member function, or declare the member to be accessed as "static".

C1803: Illegal use of address of function compiled under the pragma REG_PROTOTYPE

[ERROR]

Description

A function compiler with the pragma REG_PROTOTYPE was used in a unsafe way.

C1804: Ident expected

[ERROR]

Description

An identifier was expected.

Example

```
int ;
```

C1805: Non standard conversion used

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In ANSI-C it is normally not allowed to cast an object pointer to a function pointer or a function pointer to an object pointer. Note that the layout of a function pointer may not be the same than the layout of a object pointer.

Example

```
typedef unsigned char (*CmdPtrType)
    (unsigned char, unsigned char);
typedef struct STR {int a;} baseSTR;
```

```
baseSTR strDescriptor;
CmdPtrType myPtr;
/* next line does not strictly correspond to ANSI C
   but we make sure that the correct cast is being used */
void foo(void) {
    myPtr=(CmdPtrType)(void*)&strDescriptor; // message
C1805
}
```

C1806: Illegal cast-operation

[ERROR]

Description

There is no conversion for doing the desired cast.

Tips

The cast operator must specify a type, that can be converted to the type, which the expression containing the cast operator would be converted to.

C1807: No conversion to non-base class

[ERROR]

Description

There is no conversion from a class to another class that is not a base class of the first class.

Example

```
struct A {  
    int i;  
};  
  
struct B : A {  
    int j;  
};  
  
void main() {  
    A a;  
    B b;  
    b=a; // error: B is not a base class of A  
}
```

Tips

Remove this statement, or modify the class hierarchy.

C1808: Too many nested switch-statements

[ERROR]

Description

The number of nested switches was too high.

Example

```
switch(i0) {  
    switch(i1) {  
        switch(i2) {  
            ...  
    }
```

Tips

Use "if-else if" statements instead or reduce nesting level.

See also

[Limitations](#)

C1809: Integer value for switch-expression expected

[ERROR]

Description

The specified switch expression evaluated to an illegal type.

Example

```
float f;  
void main(void) {  
    switch(f) {  
        case 1:  
            f=2.1f;  
            break;  
        case 2:  
            f=2.1f;  
            break;  
    }  
}
```

Tips

A switch expression must evaluate to an integral type, or a class type that has an unambiguous conversion to an integral type.

C1810: Label outside of switch-statement

[ERROR]

Description

The keyword case was used outside a switch.

Tips

The keyword case can appear only within a switch statement.

C1811: Default-label twice defined

[ERROR]

Description

A switch statement must have no or one default label. Two default labels are indicated by this error.

Example

```
switch (i) {  
    default: break;  
    case 1: f(1); break;  
    case 2: f(2); break;  
    default: f(0); break;  
}
```

Tips

Define the default label only once per switch-statement.

C1812: Case-label-value already present

[ERROR]

Description

A case label value is already present.

Tips

Define a case-label-value only once for each value. Use different values for different labels.

C1813: Division by zero

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A constant expression was evaluated and found to have a zero denominator.

Example

```
int i = 1/0;
```

Tips

mod or divide should never be by zero.

Note that this error can be changed to a warning or less. This way code like the following can be compiled:

```
int i = (sizeof(int) == sizeof(long)) ? 0 : sizeof(long)
/ (sizeof(long)-sizeof(int))
```

C1814: Arithmetic or pointer-expression expected

[ERROR]

Description

The expression had an illegal type!.

Tips

Expressions after a “!” operator and expressions in conditions must be of arithmetic or pointer type.

C1815: <Name> not declared (or typename)

[ERROR]

Description

The specified identifier was not declared.

Example

```
void main(void) {  
    i=2;  
}
```

Tips

A variable's type must be specified in a declaration before it can be used. The parameters that a function uses must be specified in a declaration before the function can be used.

This error can be caused, if an include file containing the required declaration was omitted.

C1816: Unknown struct- or union-member

[ERROR]

Description

A nonmember of a structure or union was incorrectly used.

Example

```
struct A {  
    int i;  
} a;  
  
void main(void) {  
    a.I=2;  
}
```

Tips

On the right side of the “->” or “.” operator, there must be a member of the structure/union specified on the left side.

C is case sensitive.

C1817: Parameter cannot be converted to non-constant reference

[ERROR] , C++

Description

A constant argument was specified for calling a function with a reference parameter to a non-constant.

Example

```
void f(const int &); // ok
void f(int &); // causes error, when calling
                // with constant argument
void main() {
    f(3);      // error for second function declaration
}
```

Tips

The parameter must be a reference to a constant, or pass a non-constant variable as argument.

C1819: Constructor call with wrong number of arguments

[ERROR] , C++

Description

The number of arguments for the constructor call at a class object initialization was wrong.

Example

```
struct A {
    A();
};

void main() {
    A a(3); // error
}
```

Tips

Specify the correct number of arguments for calling the constructor.

Try to disable the option [_Cn=Ctr](#), so the compiler generates a copy constructor, which may be required in your code.

C1820: Destructor call must have 'void' formal parameter list

[ERROR] , C++

Description

A destructor call was specified with arguments.

Example

```
struct A {  
    ~A();  
};  
  
void main() {  
    A a;  
    a.~A(3); // error  
}
```

Tips

Destructor calls have no arguments!

C1821: Wrong number of arguments

[ERROR]

Description

A function call was specified with the wrong number of formal parameters.

Example

```
struct A {  
    void f();  
};  
  
void main() {  
    A a;  
    a.f(3);      // error  
}
```

Tips

Specify the correct number of arguments.

C1822: Type mismatch

[ERROR]

Description

There is no conversion between the two specified types.

Example

```
void main() {  
    int *p;  
    int j;  
    p=j;      // error  
}
```

Tips

Use types that can be converted.

C1823: Undefining an implicit parameter-declaration

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A implicit parameter declaration was removed because of a assignment.

Example

```
void (*f)();
void g(long );
void main(void) {
    f(1);
    f=g;
    f(2);
}
```

Tips

Avoid implicit parameter declarations whenever possible.

C1824: Indirection to different types

[ERROR]

Description

There are two pointers in the statement pointing to non-equal types.

Example

```
void main() {
    int *i;
    const int *ci;
    char *c;
    i=ci;    // C: warning, C++ error, C++ -ec warning
    i=c;    // C: warning, C++: error
}
```

Tips

Both pointers must point to equal types. If the types only differ in the qualifiers (const, volatile) try to compile with the option [-ec](#).

C1825: Indirection to different types

[DISABLE, INFORMATION, WARNING, ERROR]

Description

There are two pointers in the statement pointing to non-equal types.

Example

```
void main() {  
    int *i;  
    char *c;  
    i=c; // C: warning, C++: error  
}
```

Tips

Both pointers should point to equal types.

C1826: Integer-expression expected

[ERROR]

Description

The expression was not of integral type.

Example

```
void main() {  
    int *p;  
    p<<3;// error  
}
```

Tips

The expression must be an integral type.

C1827: Arithmetic types expected

[ERROR]

Description

After certain operators as * or /, arithmetic types must follow.

Example

```
void main() {
    int * p;
    p*3; // error
}
```

Tips

“*” and “/“ must have operands with arithmetic types.

C1828: Illegal pointer-subtraction

[ERROR]

Description

A pointer was subtracted from an integral type.

Example

```
void main() {
    int *p;
    int i;
    i-p; // error
}
```

Tips

Insert a cast operator from the pointer to the integral type.

C1829: + - incompatible Types

[ERROR]

Description

For + and - only compatible types on both sides can be used. In C++ own implementation can be used with overloading.

Example

```
struct A {  
    int i;  
};  
  
void main() {  
    int i;  
    A a;  
    i=i+a; // error  
}
```

Tips

Use compatible types on the left and on the right side of the +/- operator. Or use the operator-overloading and define an own “+” or “-” operator!

C1830: Modifiable lvalue expected

[ERROR]

Description

An attempt was made to modify an item declared with const type.

Example

```
const i;  
  
void main(void) {  
    i=2;  
}
```

Tips

Do not modify this item, or declare the item without the const qualifier.

C1831: Wrong type or not an lvalue

[ERROR]

Description

An unary operator has an operand of wrong or/and constant type.

Tips

The operand of the unary operator must be a non-const integral type or a non-const pointer to a non-void type. Or use operator overloading!.

C1832: Const object cannot get incremented

[ERROR]

Description

Constant objects can not be changed.

Example

```
int* const pi;  
void main(void) {  
    *pi++;  
}
```

Tips

Either do not declare the object as constant or use a different constant for the new value.

In the case above, use parenthesis to increment the value pi points to and to not increment pi itself.

```
int* const pi;  
void main(void) {  
    (*pi)++;  
}
```

C1833: Cannot take address of this object

[ERROR]

Description

An attempt to take the address of an object without an address was made.

Example

```
void main() {  
    register i;  
    int *p=&i; // error  
}
```

Tips

Specify the object you want to dereference in a manner that it has an address.

C1834: Indirection applied to non-pointer

[ERROR]

Description

The indirection operator (*) was applied to a non-pointer value.

Example

```
void main(void) {  
    int i;
```

```
* i=2;  
}
```

Tips

Apply the indirection operator only on pointer values.

C1835: Arithmetic operand expected

[ERROR]

Description

The unary (-) operator was used with an illegal operand type.

Example

```
const char* p= - "abc";
```

Tips

There must be an arithmetic operand for the unary (-) operator.

C1836: Integer-operand expected

[ERROR]

Description

The unary (~) operator was used with an illegal operand type.

Example

```
float f= ~1.45;
```

Tips

There must be an operand of integer type for the unary (~) operator.

C1837: Arithmetic type or pointer expected

[ERROR]

Description

The conditional expression evaluated to an illegal type.

Tips

Conditional expressions must be of arithmetic type or pointer.

C1838: Unknown object-size: sizeof (incomplete type)

[ERROR]

Description

The type of the expression in the sizeof operand is incomplete.

Example

```
int i = sizeof(struct A);
```

Tips

The type of the expression in the sizeof operand must be defined complete.

C1839: Variable of type struct or union expected

[ERROR]

Description

A variable of structure or union type was expected.

C1840: Pointer to struct or union expected

[ERROR]

Description

A pointer to a structure or union was expected.

C1842: [incompatible types

[ERROR]

Description

Binary operator “[“: There was no global operator defined, which takes the type used.

Example

```
struct A {  
    int j;  
    int operator [] (A a);  
};  
  
void main() {  
    A a;  
    int i;  
    int b[3];  
    i=a[a]; // ok  
    i=b[a]; // error  
}
```

Tips

Use a type compatible to “[“. If there is no global operator for “[“, take an integer type.

C1843: Switch-expression: integer required

[ERROR]

Description

Another type than an integer type was used in the switch expression.

Tips

Use an integer type in the switch expression.

C1844: Call-operator applied to non-function

[ERROR]

Description

A call was made to a function through an expression that did not evaluate to a function pointer.

Example

```
int i;  
void main(void) {  
    i();  
}
```

Tips

The error is probably caused by attempting to call a non-function.

In C++ classes can overload the call operator, but basic types as pointers cannot.

C1845: Constant integer-value expected

[ERROR]

Description

The case expression was not an integral constant.

Example

```
int i;  
  
void main(void) {  
    switch (i) {  
        case i+1:  
            i=1;  
    }  
}
```

Tips

Case expressions must be integral constants.

C1846: Continue outside of iteration-statement

[ERROR]

Description

A “continue” was made outside of an iteration-statement.

Tips

The “continue” must be done inside an iteration-statement.

C1847: Break outside of switch or iteration-statement

[ERROR]

Description

A “break” was made outside of an iteration-statement.

Example

```
int i;

void f(void) {
    int res;
    for (i=0; i < 10; i++ )
        res=f(-1);
    if (res == -1)
        break;
    printf("%d\n", res);
}
```

Tips

The “break” must be done inside an iteration-statement.

Check for the correct number of open braces.

C1848: Return <expression> expected

[ERROR]

Description

A return was made without an expression to be returned in a function with a non-void return type.

Tips

The return statement must return an expression of the return-type of the function.

C1849: Result returned in void-result-function

[ERROR]

Description

A return was made with an expression, though the function has void return type.

Example

```
void f(void) {  
    return 1;  
}
```

Tips

Do not return an expression in a function with void return type. Just write “return”, or write nothing.

C1850: Incompatible pointer operands

[ERROR]

Description

Pointer operands were incompatible.

Tips

Either change the source or explicitly cast the pointers.

C1851: Incompatible types

[ERROR]

Description

Two operands of a binary operator did not have compatible types (there was no conversion, or the overloaded version of the operand does not take the same types as the formal parameters).

Tips

Both operands of the binary operator must have compatible types.

C1852: Illegal sizeof operand

[ERROR]

Description

The sizeof operand was a bitfield.

Tips

Do not use bitfields as sizeof operands.

C1853: Unary minus operator applied to unsigned type

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The unary minus operator was applied to an unsigned type.

Example

```
void main(void) {  
    unsigned char c;  
    unsigned char d= -c;  
}
```

Tips

An unsigned type never can become a negative value. So using the unary minus operator may cause an unwanted behavior!

Note that ANSI C treats -1 as negated value of 1. Therefore 2147483648 is an unsigned int, if int is 32 bits large or an unsigned long if not.

The negation is a unary function as any other, so the result type is the argument type propagated to int, if smaller.

Note that the value -2147483648 is the negation of 2147483648 and therefore also of a unsigned type, even if the signed representation contains this value.

C1854: Returning address of local variable

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An address of a local variable is returned.

Example

```
int &f(void) {
    int i;
    return i;    // warning
}
```

Tips

Either change the return type of the function to the type of the local variable returned, or declare the variable to be returned as global (returning the reference of this global variable)!

C1855: Recursive function call

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A recursive function call was detected. There is a danger of endless recursion, which leads to a stack overflow.

Example

```
void f(void) {
    f();    // warning; this code leads to an endless
    recursion
}
void g(int i) {
    if(i>0) {
        g(--i); // warning; this code has no endless
        recursion
    }
}
```

```
    }  
}
```

Tips

Be sure there is no endless recursion. This would lead to a stack overflow.

C1856: Return <expression> expected

[DISABLE, ERROR, WARNING, INFORMATION], C

Description

A return statement without expression is executed while the function expects a return value. In ANSI-C, this is correct but not clean. In such a case, the program runs back to the caller. If the caller uses the value of the function call then the behavior is undefined.

Example

```
int foo(void){  
    return;  
}  
void main(void){  
int a;  
  
    ...  
    a = foo();  
    ...           /* behaviour undefined */  
}
```

Tips

```
#define ERROR_CASE_VALUE 0  
  
int foo(void){  
    return ERROR_CASE_VALUE;      /* return something... */  
}  
void main(void){  
int a;  
  
    ...  
    a = foo();
```

```
    if (a==ERROR_CASE_VALUE){ /* ... and treat this case
*/
    ...
} else {
    ...
}
...
}
```

C1857: Access out of range

[DISABLE, ERROR, WARNING, INFORMATION], C

Description

The compiler has detected that there is an array access outside of the array bounds. This is legal in ANSI-C/C++, but normally it is a programming error. Check carefully such accesses that are out of range. This warning does not check the access, but also taking the address out of an array. However, it is legal in C to take the address of one element outside the array range, so there is no warning for this. Because array accesses are treated internally like address-access operations, there is no message for accessing on element outside of the array bounds.

Example

```
char buf[3], *p;
p = &buf[3]; // no message!
buf[4] = 0; // message
```

C1858: Partial implicit parameter-declaration

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A function was called without its prototype being totally declared before.

Example

```
void foo(); // not complete prototype, arguments not
known

void main(void) {
    foo();
}
```

Tips

Prototype all arguments of the function before calling it.

C1859: Indirection operator is illegal on Pointer To Member operands

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

It is illegal to apply indirection '*' operator to Pointer To Member operands.

Example

```
class A {
public:
    void f(void) {}
};

typedef void (A::*ptrMbrFctType)(void);

void fct0(void){
    ptrMbrFctType pmf;
    *pmf=A::f; // ERROR
}

void fct1(void){
    void (* A::*pmf)(void)=A::f; // ERROR
}
```

Tips

Remove the indirection operator.

```
class A {
public:
```

```
    void f(void) {}
};

typedef void (A::*ptrMbrFctType)(void);
void fct0(void){
    ptrMbrFctType pmf;
    pmf=&A::f;
}
void fct1(void){
    void (A::*pmf)(void)=&A::f;
}
```

C1860: Pointer conversion: possible loss of data

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Whenever there is a pointer conversion which may produce loss of data, this message is produced. Loss of data can happen if a ‘far’ (e.g. 3 byte pointer) is assigned to a pointer of smaller type (e.g. a ‘near’ 1 byte pointer).

Example

```
char *near nP;
char *far fP;
void foo(void) {
    nP = fP; // warning here
}
```

Tips

Check if this loss of data is intended.

C1861: Illegal use of type ‘void’

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler has detected an illegal usage of the ‘void’ type. The compiler accepts this because of historical reasons. Some other vendor compilers may not accept this at all, so this may cause portability problems.

Example

```
int foo(void buf[256]) { // warning here  
    ...  
}
```

Tips

Correct your code. E.g. replace in the above example the argument with ‘void *buf’.

C2000: No constructor available

[ERROR] , C++

Description

A constructor must be called, but none is available.

Tips

Define a constructor. No compiler defined default constructor is defined in some situations, for example when the class has constant members.

C2001: Illegal type assigned to reference.

[ERROR] , C++

Description

There is no conversion from type assigned to the type of the reference.

Example

```
int *i;  
int &r=i; // error
```

Tips

The type of the reference must be equal to the type assigned.

C2004: Non-volatile reference initialization with volatile illegal

[ERROR] , C++

Description

The reference type is not volatile, the assigned type is.

Example

```
volatile i;  
const int &r=i; // illegal
```

Tips

Either both are volatile or both are not volatile.

C2005: Non-constant reference initialization with constant illegal

[ERROR] , C++

Description

The reference type is not constant, the assigned type is.

Example

```
void main(void) {  
    const int i=1;  
    int &p=i;  
}
```

Tips

Either both are const or both are not const.

C2006: (un)signed char reference must be const for init with char

[ERROR] , C++

Description

The initializer for a reference to a signed or unsigned char must be const for initialization with a plain char.

Example

```
char i;  
signed char &r=i; // error
```

Tips

Either declare the reference type as const, or the type of the initializer must not be plain.

C2007: Cannot create temporary for reference in class/struct

[ERROR] , C++

Description

A member initializer for a reference was constant, though the member was non-constant

Example

```
struct A {  
    int &i;
```

```
        A( );
    };
A::A() : i(3) { // error
}
```

Tips

Initialize the reference with a non-constant variable.

C2008: Too many arguments for member initialization

[ERROR], C++

Description

The member-initializer contains too many arguments.

Example

```
struct A {
    const int i;
    A();
};
A::A() : i(3,5) { // error
}
```

Tips

Supply the correct number of arguments in the initializer list of a constructor.

C2009: No call target found!

[ERROR]

Description

The ambiguity resolution mechanism did not find a function in the scope, where it expected one.

Tips

Check, if the function called is declared in the correct scope.

C2010: <Name> is ambiguous

[ERROR], C++

Description

The ambiguity resolution mechanism found an ambiguity. That means, more than one object could be taken for the identifier Name. So the compiler does not know which one is desired.

Example

```
struct A {  
    int i;  
};  
struct B : A {  
};  
struct C : A {  
};  
struct D : B, C {  
};  
void main() {  
    D d;  
    d.i=3;    // error, could take i from B::A or C::A  
    d.B::i=4; // ok  
    d.C::i=5; // ok  
}
```

Tips

Specify a path, how to get to the desired object. Or use virtual base classes in multiple inheritance.

The compiler can handle a most 10'000 different numbers for a compilation unit. Internally for each number a descriptor exists. If an internal number descriptor already exists for a given number value with a given type, the existing one is

used. But if e.g. more than 10'000 different numbers are used, this message will appear.

C2011: <Name> can not be accessed

[ERROR] , C++

Description

There is no access to the object specified by the identifier.

Example

```
struct A {  
private:  
    int i;  
protected:  
    int j;  
public:  
    int k;  
    void g();  
};  
  
struct B : public A {  
    void h();  
};  
  
void A::g() {  
    this->i=3; // ok  
    this->j=4; // ok  
    this->k=5; // ok  
}  
  
void B::h() {  
    this->i=3; // error  
    this->j=4; // ok  
    this->k=5; // ok  
}  
  
void f() {  
    A a;  
    a.i=3; // error  
    a.j=4; // error  
    a.k=5; // ok  
}
```

Tips

Change the access specifiers in the class declaration, if you really need access to the object.

Or use the friend-mechanism.

See also

Limitations

C2012: Only exact match allowed yet or ambiguous!

[ERROR], C++

Description

An overloaded function was called with non-exact matching arguments.

Tips

Supply exact matching parameters.

In later compiler versions this error will disappear!

See also

Limitations

C2013: No access to special member of base class

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

The special member (Constructor, destructor or assignment operator) could not be accessed.

Example

```
struct A {  
    private:  
        A();  
};  
struct B : A {  
};  
void main() {  
    B b; // error  
}
```

Tips

Change the access specifier for the special member.

See also

[Limitations](#)

C2014: No access to special member of member class

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

The special member (Constructor, destructor or assignment operator) could not be accessed.

Example

```
struct A {  
    private:  
        A();  
};  
struct B {  
    A a;  
};  
void main() {  
    B b; // error  
}
```

Tips

Change the access specifier for the special member.

See also

Limitations

C2015: Template is used with the wrong number of arguments

[ERROR], C++

Description

A template was instantiated with the wrong number of template arguments.

Example

```
template<class S> struct A {  
    S s;  
};  
A<int, int> a; // error
```

Tips

The instantiation of a template type must have the same number of parameters as the template specification.

C2016: Wrong type of template argument

[ERROR], C++

Description

A template was instantiated with the wrong type template arguments.

Example

```
template<class S> struct A {  
    S s;  
};  
A<4> a; // error
```

Tips

The instantiation of a template type must have the same argument type as the ones template specification.

C2017: Use of incomplete template class

[ERROR], C++

Description

A template was instantiated from an incomplete class.

Example

```
template<class S> struct A;  
A<int, int> a; // error
```

Tips

The template to be instantiated must be of a defined class.

C2018: Generate class/struct from template

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

A class was instantiated from a template.

Example

```
template<class S> struct A {  
    S s;  
};  
  
A<int> a; // information
```

C2019: Generate function from template

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

A function was instantiated from a template.

Example

```
template<class S> void f(S s) {  
    /* ... */  
}  
  
void main(void) {  
    int i;  
    char ch;  
    f(4); // generate  
    f(i); // not generating, already generated  
    f(ch); // generate  
}
```

Tips

The fewer functions are instantiated from a template, the less code is produced.
So try to use already generated template functions instead of letting the compiler
generate new ones.

C2020: Template parameter not used in function parameter list

[ERROR], C++

Description

A template parameter didn't occur in the parameter list of the template function.

Example

```
template<class S> void f(int i) { // error
    /* ... */
}
```

Tips

The parameter list of the template function must contain the template parameters.

C2021: Generate NULL-check for class pointer

[DISABLE, INFORMATION, WARNING, ERROR], C++

Description

Operations with pointers to related classes always need separate NULL-checks before adding offsets from base classes to inherited classes.

Example

```
class A {
};

class B : public A{
};

void main() {
    A *ap;
    B *bp;
    ap=bp; // warning
}
```

Tips

Try to avoid operations with pointers to different, but related classes.

C2022: Pure virtual can be called only using explicit scope resolution

[ERROR] , C++

Description

Pure virtual functions can be called from a constructor of an abstract class; the effect of calling a pure virtual function directly or indirectly for the object being created from such a constructor is an error.

Example

```
class A{
public:
    virtual void f(void) = 0;
    A(){
        f();
    }
};
```

Tips

A pure virtual can be defined. It can be called using explicit qualification only.

```
class A{
public:
    virtual void f(void) = 0;
    A(){
        A::f();
    }
};
void A::f(void){ // defined somewhere
    //...
}
```

C2023: Missing default parameter

[ERROR], C++

Description

A subsequent parameter of a default parameter is not a default parameter.

Example

```
void f(int i=0, int j);    // error
void f(int i=0, int j=0); // ok
```

Tips

All subsequent parameters of a default parameter must be default, too.

See also

Overloading.

C2024: Overloaded operators cannot have default arguments

[ERROR], C++

Description

An overloaded operator was specified with default arguments.

Example

```
struct A{ /* ... */ };
operator + (A a, int i=0); // error
```

Tips

Overloaded operators cannot have default arguments. Declare several versions of the operator with different numbers of arguments.

See also

Overloading.

C2025: Default argument expression can only contain static or global objects or constants

[ERROR] , C++

Description

A local object or non-static class member was used in the expression for a default argument.

Example

```
struct A {  
    int t;  
    void f(int i=t); // error  
};  
void g(int i, int j=i); // error
```

Tips

Only use static or global objects or constants in expressions for default arguments.

C2200: Reference object type must be const

[ERROR] , C++

Description

If a reference is initialized by a constant, the reference has to be constant as well

Example

```
int &ref = 4; // err
```

Tips

Declare the reference as constant.

See also

[Limitations](#)

C2201: Initializers have too many dimensions

[ERROR], C++

Description

An initialization of an array of class objects with constructor call arguments was having more opening braces than dimensions of the array.

Example

```
struct A {  
    A(int);  
};  
  
void main() {  
    A a[3]={ {3,4}, 4}; // errors  
    A a[3]={3,4,4};    // ok  
}
```

Tips

Provide the same number of opening braces in an initialization of an array of class objects.

See also

[Limitations](#)

C2202: Too many initializers for global Ctor arguments

[ERROR], C++

Description

An initialization of a global array of class objects with constructor call arguments was having more arguments than elements in the array.

Example

```
struct A {  
    A(int);  
};  
  
A a[3]={3,4,5,6}; // errors  
  
A a[3]={3,4,5}; // ok
```

Tips

Provide the same number of arguments than number of elements in the global array of class objects.

If you want to make calls to constructors with more than one argument, use explicit calls of constructors.

See also

Limitations

C2203: Too many initializers for Ctor arguments

[ERROR], C++

Description

An initialization of an array of class objects with constructor call arguments was having more arguments than elements in the array.

Example

```
struct A {  
    A(int);  
};  
  
void main() {  
    A a[3]={3,4,5,6}; // errors  
    A a[3]={3,4,5}; // ok  
}
```

Tips

Provide the same number of arguments than number of elements in the array of class objects.

If you want to make calls to constructors with more than one argument, use explicit calls of constructors.

Example

```
struct A {  
    A(int);  
    A();  
    A(int,int);  
};  
  
void main() {  
    A a[3]={A(3,4),5,A()};  
    // first element calls A::A(int,int)  
    // second element calls A::A(int)  
    // third element calls A::A()  
}
```

See also

[Limitations](#)

C2204: Illegal reinitialization

[ERROR]

Description

The variable was initialized more than once.

Example

```
extern int i=2;  
  
int i=3; // error
```

Tips

A variable must be initialized at most once.

See also

[Limitations](#)

C2205: Incomplete struct/union, object can not be initialized

[ERROR]

Description

Incomplete struct/union, object can not be initialized

Example

```
struct A;  
extern A a={3}; // error
```

Tips

Do not initialize incomplete struct/union. Declare first the struct/union, then initialize it.

See also

[Limitations](#)

C2206: Illegal initialization of aggregate type

[ERROR]

Description

An aggregate type (array or structure/class) was initialized the wrong way.

Tips

Use the braces correctly.

See also

Limitations

C2207: Initializer must be constant

[ERROR]

Description

A global variable was initialized with a non-constant.

Example

```
int i;  
  
int j=i; // error  
  
or  
  
void function(void){  
    int local;  
    static int *ptr = &local;  
    // error: address of local can be different  
    // in each function call.  
    // At second call of this function *ptr is not the  
    same!  
}
```

Tips

In C, global variables can only be initialized by constants. If you need non-constant initialization values for your global variables, create an “`InitModule()`” function in your compilation unit, where you can assign any expression to your globals. This function should be called at the beginning of the execution of your program.

If you compile your code with C++, this error won’t occur anymore!

In C, initialization of a static variables is done only once. Initializer is not required to be constant by ANSI-C, but this behavior will avoid troubles hard to debug.

See also

[Limitations](#)

C2209: Illegal reference initialization

[ERROR], C++

Description

A reference was initialized with a braced “{“, “}” initializer.

Example

```
struct A {  
    int i;  
};  
A &ref = {4}; // error  
A a = {4}; // ok  
A &ref2 = a; // ok
```

Tips

References must be initialized with non-braced expressions.

See also

[Limitations](#)

C2210: Illegal initialization of non-aggregate type

[ERROR]

Description

A class without a constructor and with non-public members was initialized.

Tips

Classes with non-public members can only be initialized by constructors.

See also

Limitations

C2211: Initialization of a function

[ERROR]

Description

A function was initialized.

Example

```
void f( )=3;
```

Tips

Functions cannot be initialized. But function pointers can.

See also

Limitations

C2401: Pragma <ident> expected

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A single #pragma was found.

Example

```
#pragma
```

Tips

Probably this is a bug. Correct it.

C2402: Variable <Ident> <State>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A variable was allocated differently because of a pragma INTO_ROM or a pragma FAR.

Example

```
#pragma INTO_ROM  
const int i;
```

Tips

Be careful with the pragmas INTO_ROM and pragma FAR. They are only valid for one single variable. In the following code the pragma INTO_ROM puts var_rom into the rom, but var_ram not.

```
#pragma INTO_ROM  
const int var_rom, var_ram;
```

Note that pragma INTO_ROM is only for the HIWARE Object file format.

C2450: Expected: <list of expected keywords and tokens>

[ERROR]

Description

An unexpected token was found.

Example

```
void f(void);  
  
void main(void) {  
    int i=f(void); // error: "void" is an unexpected  
    keyword!  
}
```

Tips

Use a token listed in the error message. Check if you are using the right compiler language option. E.g. you may compile a file with C++ keywords, but are not compiling the file with C++ option set.

Too many nested scopes

C2550: Too many nested scopes

[FATAL]

Description

Too many scopes are open at the same time.

For the actual limitation number, please see chapter [Limitations](#)

Example

```
void main(void) {  
    {  
        {  
            {  
                ....
```

Tips

Use less scopes.

See also

[Limitations](#)

C2700: Too many numbers

[FATAL]

Description

Too many different numbers were used in one compilation unit.

For the actual limitation number, please see chapter [Limitations](#)

Example

```
int main(void) {  
    return 1+2+3+4+5+6+.....  
}
```

Tips

Split up very large compilation units.

See also

[Limitations](#)

C2701: Illegal floating-point number

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An illegal floating point number has been specified or the exponent specified is to large for floating number.

Example

```
float f = 3.e345689;
```

Tips

Correct the floating point number.

See also

Number Formats
header file “float.h”

C2702: Number too large for float

[ERROR]

Description

A float number larger than the maximum value for a float has been specified.

Example

```
float f = 3.402823466E+300F;
```

Tips

Correct the number.

See also

Number Formats
header file “float.h”

C2703: Illegal character in float number

[ERROR]

Description

The floating number contains an illegal character. Legal characters in floating numbers are the postfixes ‘f’ and ‘F’ (for float) or ‘l’ and ‘L’ (for long double). Valid characters for exponential numbers are ‘e’ and ‘E’.

Example

```
float f = 3.x4;
```

Tips

Correct the number.

See also

[Number Formats](#)

C2704: Illegal number

[ERROR]

Description

An illegal immediate number has been specified.

Example

```
int i = 4x; /* error */  
float f= 12345678901234567890;//error too large for a  
long!  
float f= 12345678901234567890.;//OK, doubles can be as  
large
```

Tips

Correct the number.

For floating point numbers, specify a dot.

See also

[Number Formats](#)

C2705: Possible loss of data

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler generates this message if a constant is used which exceeds the value for a type.

Another reason for this message is if a object (e.g. long) is assigned to an object with smaller size (e.g. char).

Another example is to pass an actual argument too large for a given formal argument, e.g. passing a 32bit value to a function which expects a 8bit value.

Example

```
signed char ch = 128; /* char holds only -128..127 */
char c;
long L;

void foo(short);

void main(void) {
    c = L; // possible lost of data
    foo(L); // possible lost of data
}
```

Tips

Usually this is a programming error.

See also

Header file "limits.h"

C2706: Octal Number

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An octal number was parsed.

Example

```
int f(void) {
    return 0100; // warning
}
```

Tips

If you want to have a decimal number, don't write a '0' at the beginning.

C2707: Number too large

[ERROR] , M2

Description

While reading a numerical constant, the compiler has detected the number is too large for a data type.

Example

```
x: REAL;  
x := 300e51234;
```

Tips

Reduce the numerical constant value, or choose another data type.

C2708: Illegal digit

[ERROR] , M2

Description

While reading a numerical constant, an illegal digit has been found.

Example

```
x: REAL;  
x := 123e4a;
```

Tips

Check your numerical constant for correctness.

C2709: Illegal floating-point exponent ('-', '+' or digit expected)

[ERROR]

Description

While reading a numerical constant, an illegal exponent has been found.

Example

```
x = 123e;
```

Tips

Check your numerical constant for correctness. After the exponent, there has to be an optional '+' or '-' sign followed by a sequence of digits.

C2800: Illegal operator

[ERROR]

Description

An illegal operator has been found. This could be caused by an illegal usage of saturation operators, e.g. the using saturation operators without enabling them with a compiler switch if available.

Note that not all compiler backends support saturation operators.

Example

```
i = j +? 3; /* illegal usage of saturation operator */
```

Tips

Enable the usage of Saturation operators if available.

See also

Saturation Operator

Compiler Backend Chapter

C2801: <Symbol> missing"

[ERROR]

Description

There is a missing symbol for the Compiler to complete a parsing rule. Normally this is just a closing parenthesis or a missing semicolon.

Example

```
void main(void) {  
    // '}' missing  
other Example  
void f() {  
    int i  
}
```

Tips

Usually this is a programming error. Correct your source code!

C2802: Illegal character found: <Character>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In the source there is a character which does not match with the name rules for C/C++. As an example it is not legal to have '\$' in identifiers.

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Example

```
int $j;
```

Tips

Usually this is a programming error. Replace the illegal character with a legal one. Some E-MAIL programs set the most significant bit of two immediately following spaces. In a hex editor, such files then contain “a0 a0 a0 20” for four spaces instead of “20 20 20 20”. When this occurs in your E-Mail configuration, send sources as attachment.

C2803: Limitation: Parser was going out of sync!

[ERROR]

Description

The parser was going out of synchronization.

This is caused by complex code with many blocks, gotos and labels.

Example

It would take too much space to write an example here!

Tips

Try to simplify your code!

See also

Limitations

C2900: Constant condition found, removing loop

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A constant loop condition has been found and the loop is never executed. No code is produced for such a loop. Normally, such a constant loop condition may be a programming error.

Example of a constant loop condition:

```
for(i=10; i<9; i--)
```

Because the loop condition ‘i<9’ never becomes true, the loop is removed by the compiler and only the code for the initialization ‘i=10’ is generated.

Tips

If it is a programming error, correct the loop condition.

See also

Loop Unrolling

C2901: Unrolling loop

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A loop has been detected for unrolling. Either the loop has only one round, e.g.

```
for(i=1; i<2; i++)
```

or loop unrolling was explicitly requested (Compiler Option [-Cu](#) or pragma LOOP_UNROLL) or the loop has been detected as empty as

```
for(i=1; i<200; i++);
```

Tips

If it is a programming error, correct the loop condition.

See also

Loop Unrolling

C3000: File-stack-overflow (recursive include?)

[FATAL]

Description

There are more than 256 file includes in one chain or a possible recursion during an include sequence. Maybe the included header files are not guarded with `#ifndef`

Example

```
/* foo.c */
#include "foo.c"
```

Tips

Use `#ifndef` to break a possible recursion during include:

```
/* foo.h */
#ifndef FOO_H
#define FOO_H

/* additional includes, declarations, ... */

#endif
```

Simplify the include complexity to less than 256 include files in one include chain.

See also

Limitations

C3100: Flag stack overflow -- flag ignored

[DISABLE, INFORMATION, WARNING, ERROR], MODULA-2

Description

There were too many flags used at the same time. This message occurs for Modula 2 versions of the compiler only. It occurs for C/C++ compilers.

C3200: Source file too big

[FATAL]

Description

The compiler can handle about 400'000 lexical source tokens. A source token is either a number or an ident, e.g. ‘int a[2] = {0,1};’ contains the 12 tokens ‘int’, ‘a’, ‘[’, ‘2’, ‘]’, ‘=’, ‘{’, ‘0’, ‘1’, ‘}’ and ‘;’.

Example

A source file with more than 400'000 lexical tokens.

Tips

Split up the source file into parts with less than 400'000 lexical tokens.

See also

Limitations

C3201: Carriage-Return without a Line-Feed was detected

[DISABLE, INFORMATION, WARNING, ERROR]

Description

On a PC, the usual ‘newline’ sequence is a carriage return (CR) followed by a line feed (LF). With this message the compiler warns that there is a CR without a LF. The reason could be a not correctly transformed UNIX source file. However, the compiler can handle correct UNIX source files (LF only).

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Tips

Maybe the source file is corrupted or the source file is not properly converted from a UNIX source file. Try to load the source file into an editor and to save the source file, because most of the editors will correct this.

C3202: Ident too long

[FATAL]

Description

An identifier was longer than allowed. The compiler supports identifiers with up to 16000 characters. Note that the limits of the linker/debugger may be smaller.

The 16000 characters are in respect of the length of the identifier in the source. A name mangled C++ identifier is only limited by available memory.

Tips

Do not use such extremely large names!

C3300: String buffer overflow

[FATAL]

Description

The compiler can handle a maximum of about 10'000 strings in a compilation unit. If the compilation unit contains too many strings, this message will appear.

Example

A source file with more than 10'000 strings, e.g.

```
char *chPtr[] = {"string0", "string1",
                  "string2", ... "string1000"};
```

Tips

Split up the source file into parts with less than 10'000 strings.

See also

[Limitations](#)

C3301: Catenated string too long

[FATAL]

Description

The compiler cannot handle an implicit string concatenation with a total of more than 8192 characters.

Example

Implicit string concatenation of two strings with each more than 4096 characters:

```
char *str = "MoreThan4096...."  
           "OtherWithMoreThan4096...."
```

Tips

Do not use implicit string concatenation, write the string in one piece:

```
char *str = "MoreThan4096....OtherWithMoreThan4096...."
```

See also

[Limitations](#)

[Message C3303](#)

C3302: Prae_number-buffer overflow

[FATAL]

Description

This message may occur during preprocessing if there are too many numbers to handle for the compiler in a compilation unit.

The compiler can handle a most 10'000 different numbers for a compilation unit. Internally for each number a descriptor exists. If an internal number descriptor already exists for a given number value with a given type, the existing one is used. But if e.g. more than 10'000 different numbers are used, this message will appear.

Example

An array initialized with the full range of numbers from 0 to 10'000:

```
const int array[ ] = { 0, 1, 2, ... 10000 };
```

Tips

Splitting up the source file into smaller parts until this message disappears.

See also

[Limitations](#)

C3303: Implicit concatenation of strings

[DISABLE, INFORMATION, WARNING, ERROR]

Description

ANSI-C allows the implicit concatenation of strings: Two strings are merged by the preprocessor if there is no other preprocessor token between them. This is a useful feature if two long strings should be one entity and you do want to write a very long line:

```
char *message = "This is my very long message string
which "
"which has been splitted into two parts!"
```

This feature may be dangerous if there is just a missing comma (see example below!). If intention was to allocate a array of char pointers with two elements, the compiler only will allocate one pointer to the string "abcdef" instead two pointers if there is a comma between the two strings.

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Example

```
char *str[ ] = { "abc" "def" }; /* same as "abcdef" */
```

Tips

If it is a programming error, correct it.

C3304: Too many internal ids, split up compilation unit

[FATAL]

Description

The compiler internally maintains some internal id's for artificial local variables.
The number of such internal id's is limited to 256 for a single compilation unit.

Tips

Split up the compilation unit.

See also

Limitations

C3400: Cannot initialize object (dest too small)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An object cannot been initialized, because the destination is too small, e.g. because the pointer is too small to hold the address. The message typically occurs if the programmer tries to initialize a ‘near’ pointer (e.g. 16bit) with a ‘far’ pointer (e.g. 24bit).

Example

```
#pragma DATA_SEG FAR MyFarSegment  
char Array[10];  
  
#pragma DATA_SEG DEFAULT  
char *p = Array;
```

Tips

Increase the type size for the destination (e.g. with using the ‘far’ keyword if supported)

```
char *far p = Array;
```

Be careful when disabling this message.

See also

Limitations

C3401: Resulting string is not zero terminated

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler issues this message if a the resulting string is not terminated with a zero byte. Thus if such a string is used for printf or strcpy, the operation may fail. In C it is legal to initialize an array with a string if the string fits without the zero byte.

Example

```
void main(void) {  
    char buf[3] = "abc";  
}
```

Tips

For array initialization it is always better to use [] instead to specify the size:

```
char buf[] = "abc";
```

C3500: Not supported fixup-type for ELF-Output occurred

[FATAL]

Description

A fixup type not supported by the ELF Object file writer occurred. This message indicates an internal compiler error because all necessary fixup types must be supported. This message also can occur if in HLI (High Level Inline) Assembler an unsupported relocation/fixup type is used.

Tips

Report this error to your distributor.

C3501: ELF Error <Description>

[FATAL]

Description

The ELF generation module reports an error. Possible causes are when the object file to be generated is locked by another application or the disk is full.

Tips

Check if the output file exists and is locked. Close all applications which might lock it.

C3600: Function has no code: remove it!

[ERROR]

Description

It is an error when a function is really empty because such a function can not be represented in the memory and it does not have an address.

Because all C functions have at least a return instruction, this error can only occur with the pragma NO_EXIT. Remark that not all targets support NO_EXIT.

Example

```
#pragma NO_EXIT  
void main(void) { }
```

Tips

Remove the function. It is not possible to use an empty function.

C3601: Pragma TEST_CODE: mode <Mode>, size given <Size> expected <Size>, hashcode given <HashCode>, expected <HashCode>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The condition tested with a pragma TEST_CODE was false.

Example

```
#pragma TEST_CODE == 0
void main(void) {}
```

Tips

There are many reasons why the generated code may have been changed.
Check why the pragma was added and which code is now generated.
If the code is correct, adapt the pragma. Otherwise change the code.

See also

#pragma TEST_CODE description

C3602: Global objects: <Number>, Data Size (RAM): <Size>, Const Data Size (ROM): <Size>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is used to give some additional information about the last compilation unit compiled.

Example

Compile any C file.

Tips

Smart Linking may not link all variables or constants, so the real application may be smaller than this value.

No alignment bytes are counted.

C3603: Static '<Function>' was not defined

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A static function was used, but not defined.

As static functions can only be defined in this compilation unit, the function using the undefined static cannot successfully link.

Example

```
static void f(void);  
void main(void) {  
    f();  
}
```

Tips

Define the static function, remove its usage or declare it as external.

C3604: Static '<Object>' was not referenced

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A static object was defined but not referenced.

When using smart linking, this object will not be used.

Example

```
static int i;
```

Tips

Remove the static object, comment it out or do not compile it with conditional compilation. Not referenced static functions often exist because they were used sometime ago but no longer or because the usage is present but not compiled because of conditional compilation.

C3605: Runtime object '<Object>' is used at PC <PC>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

By default this message is disabled. This message may be enabled to report every runtime object used. Runtime objects (or calls) are used if the target CPU itself does not support a specific operation, e.g. a 32bit division or if the usage of such a runtime function is better than directly to inline it.

The message is issued at the end of a function and reports the name and the PC where the object is used.

Example

```
double d1, d2;  
  
void foo(void) {  
    d1 = d2 * 4.5; // compiler calls _DMUL for  
                   // IEEE64 multiplication  
}
```

Tips

Set this message to an error if you have to ensure that no runtime calls are made.

C3606: Initializing object '<Object>'

[DISABLE, INFORMATION, WARNING, ERROR]

Description

If global variables are initialized, such initialized objects have to be initialized during startup of the application. This message (which is disabled by default) reports every single global or static local initialization.

Example

```
int i = 3;           // message C3606
char *p = "abc";    // message C3606
void foo(void) {
    int k = 3;       // no message!
    static int j = 3; // message C3606
}
```

Tips

Set this message to an error if you have to ensure that no additional global initialization is necessary for a copy down during startup of the application.

C3700: Special opcode too large

[FATAL]

Description

An internal buffer overflow in the ELF/DWARF 2 debug information output. This error indicates an internal error. It should not be possible to generate this error with legal input.

C3701: Too many attributes for DWARF2.0 Output

[FATAL]

Description

The ELF/DWARF 2 debug information output supports 128*128-128 different DWARF tags. This error indicates an internal error. It should not be possible to

generate this error with legal input because similar objects use the same tag and there much less possible combinations.

C3800: Segment name already used

[ERROR]

Description

The same segment name was used for different segment type.

Example

```
#pragma DATA_SEG Test  
#pragma CODE_SEG Test
```

Tips

Use different names for different types. If the two segments must be linked to the same area, this could be done in the link parameter file.

C3801: Segment already used with different attributes

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A segment was used several times with different attributes.

Example

```
#pragma DATA_SEG FAR AA  
..  
#pragma DATA_SEG NEAR BB  
..
```

Tips

Use the same attributes with one segment. Keep variables of the same segment together to avoid inconsistencies.

C3802: Segment pragma incorrect

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A section pragma was used with incorrect attributes or an incorrect name.

Example

```
#pragma DATA_SEG FAR FAR
```

Tips

Take care about not using keywords as names segment names. Note that you can use e.g. the __FAR_SEG instead FAR.

Example

```
#pragma DATA_SEG __FAR_SEG MyFarSeg
```

C3803: Illegal Segment Attribute

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A Segment attribute was recognized, but this attribute is not applicable to this segment.

Code segments may only be FAR, NEAR and SHORT. The DIRECT attribute is allowed for data segments only.

The actual available segment attributes and their semantic depends on the target processor.

Example

```
#pragma CODE_SEG DIRECT MyFarSegment
```

Tips

Correct the attribute. Do not use segment attribute specifiers as segment names.
Note that you can use the ‘safe’ qualifiers as well, e.g. __FAR_SEG.

C3804: Predefined segment '<segmentName>' used

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A Segment name was recognized which is a predefined one. Predefined segment names are FUNCS, STRINGS, ROM_VAR, COPY, STARTUP, _PRESTART, SSTACK, DEFAULT_RAM, DEFAULT_ROM and _OVERLAP.

If you use such segment names, this may raise conflicts during linking.

Example

```
#pragma CODE_SEG FUNCS // WARNING here
```

Tips

Use another name. Do not use predefined segment names.

C3900: Return value too large

[ERROR]

Description

The return type of the function is too large for this compiler.

Example

```
typedef struct A {  
    int i,j,k,l,m,n,o,p;  
}A;  
  
A f();
```

Tips

In C++, instead of a class/struct type, return a reference to it! In C, allocate the structure at the caller and pass a pointer/reference as additional parameter.

See also

[Compiler Backend](#)

C4000: Condition always is TRUE

[DISABLE, [INFORMATION](#), WARNING, ERROR]

Description

The compiler has detected a condition to be always true. This may also happen if the compiler uses high level optimizations, but could also be a hint for a possible programming error.

Example

```
unsigned int i;  
  
if (i >= 0) i = 1;
```

Tips

If it is a programming error, correct the statement

C4001: Condition always is FALSE

[DISABLE, [INFORMATION](#), WARNING, ERROR]

Description

The compiler has detected a condition to be always false. This may also happen if the compiler uses high level optimizations, but could also be a hint for a possible programming error.

Example

```
unsigned int i;  
if (-i < 0) i = -i;
```

Tips

If it is a programming error, correct the statement

C4002: Result not used

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The result of an expression outside a condition is not used. In ANSI-C it is legal to write code as in the example below. Some programmers are using such a statement to enforce only a read access to a variable without write access, but in most cases the compiler will optimize such statements away.

Example

```
int i;  
i+1; /* should be 'i=1;', but programming error */
```

Tips

If it is a programming error, correct the statement.

C4003: Shift count converted to unsigned char

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In ANSI-C, if a shift count exceeds the number of bits of the value to be shifted, the result is undefined. Because there is no integral data type available with more than 256 bits yet, the compiler implicitly converts a shift count larger than 8 bits (char) to an unsigned char, avoiding loading a shift count too large for shifting, which does not affect the result.

This ensures that the code is as compact as possible.

Example

```
int j, i;  
i <= j; /* same as 'i <= (unsigned char)j;' */
```

In the above example, both ‘i’ and ‘j’ have type ‘int’, but the compiler can safely replace the ‘int’ shift count ‘j’ with a ‘unsigned char’ type.

Tips

None, because it is a hint of compiler optimizations.

C4004: BitSet/BitClr bit number converted to unsigned char

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler has detected a shift count larger than 8bit used for a bitset/bitclear operation. Because it makes no sense to use a shift count larger than 256, the compiler optimizes the shift count to a character type. Reducing the shift count may reduce the code size and improve the code speed (e.g. a 32bit shift compared with a 8bit shift).

Example

```
int j; long L;  
j |= (1<<L); // the compiler converts 'L'  
// to a unsigned character type
```

Tips

None, because it is a hint of compiler optimizations.

C4005: Expression with a cast is not a lvalue

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In ANSI-C, an expression which modifies a value has to be a modifiable lvalue. If the expression has a cast as root, the expression itself is not a lvalue.

Example

```
int i;  
(int)i++;
```

Tips

Remove the cast if it is useless.

C4006: Expression too complex

[FATAL]

Description

The compiler cannot handle an expression which has more than 32 recursion levels.

Example

```
typedef struct S {  
    struct S *n;  
} S;  
S *s;  
  
void foo(void) {  
    s->n->n->n->n-> ... n->n->n->n->n->n->n = 0;  
}
```

Tips

Try to simplify the expression, e.g. use temporary variables to hold expression results.

See also

[Limitations](#)

C4100: Converted bit field signed -1 to 1 in comparison

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A signed bitfield entry of size 1 can only have the values 0 and -1. The compiler did find a comparison of such a value with 1. The compiler did use -1 instead to generate the expected code.

Example

```
struct A {
    int i:1;
} a;

void f(void);

void main(void) {
    if (a.i == 1) {
        f();
    }
}
```

Tips

Correct the source code. Either use an unsigned bitfield entry or compare the value to -1.

C4101: Address of bitfield is illegal

[ERROR]

Description

The address of a bitfield was taken.

Example

```
typedef struct A {
    int bf1:1;
} A;

void f() {
    A a;
    if(&a.bf1);
}
```

Tips

Use a "normal" integral member type, if you really need to have the address.

C4200: Other segment than in previous declaration

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A object (variable or function) was declared with inconsistent segments.

Example

```
#pragma DATA_SEG A
extern int i;

#pragma DATA_SEG B
int i;
```

Tips

Change the segment pragmas so, that all declarations and the definition of one object are in the same segment. Otherwise wrong optimizations could happen.

C4201: pragma <name> was not handled

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A pragma was not used by the compiler. This may have different reasons:

- the pragma is intended for a different compiler
- by a typing mistake, the compiler did not recognize a pragma. Note that pragma names are case sensitive.
- there was no context, a specific pragma could take some action
- The segment pragmas DATA_SEG, CODE_SEG, CONST_SEG and their aliases never issue this warning, even if they are not used.

Example

```
#pragma TRAP_PROG
// typing mistake: the interrupt pragma is called
TRAP_PROC

void Inter(void) {
    ...
}
```

Tips

Investigate this warning carefully.

This warning can be mapped to an error if only pragmas are used which are known.

C4202: Invalid pragma OPTION, <description>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A ill formed pragma OPTION was found or the given options were not valid. The description says more precisely what is the problem with a specific [pragma OPTION](#).

Example

```
#pragma OPTION add "-or"
```

The pragma OPTION keyword ADD is case sensitive!

Write instead:

```
#pragma OPTION ADD "-or"
```

Tips

When the format was illegal, correct it. You can add comments, but they must follow the usual C rules.

Be careful which options are given by the command line when adding options.

It is not possible to add options which contradicts to command line options.

Notice the limitations of the [pragma OPTION](#).

See also

[pragma OPTION](#)

C4203: Invalid pragma MESSAGE, <description>

[DISABLE, INFORMATION, WARNING, [ERROR](#)]

Description

A ill formed pragma MESSAGE was found or the given message number cannot be moved. The description says more precisely what is the problem with a specific [pragma MESSAGE](#).

Example

```
#pragma MESSAGE warning C4203
```

The pragma OPTION keyword warning is case sensitive!

Write instead:

```
#pragma MESSAGE WARNING C4203
```

Tips

When the format was illegal, correct it. You can add comments, but they must follow the usual C rules.

The same message can be moved at different code positions to a different state.

Be careful not to specify the same message with a option or with graphical user interface and with this pragma. If this is done, it is not defined which definition is actually taken.

See also

[pragma MESSAGE](#)

C4204: Invalid pragma REALLOC_OBJ, <description>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The pragma REALLOC_OBJ was used in a ill formed way. The correct syntax this pragma is can be found here: [#pragma REALLOC_OBJ](#)

See also

[#pragma REALLOC_OBJ](#)

Linker Manual

C4205: Invalid pragma LINK_INFO, <description>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The pragma LINK_INFO was used in a ill formed way. The correct syntax this pragma is can be found here: [#pragma LINK_INFO](#)

See also

[#pragma LINK_INFO](#)

C4300: Call of an empty function removed

[DISABLE, INFORMATION, WARNING, ERROR]

Description

If the option [-Oi](#) is enabled, calls of empty functions are removed.

Example

```
void f() {  
}  
void main() {  
    f();           // this call is removed !  
}
```

Tips

If for any reason you need a call of an empty function, disable the option [-Oi](#).

See also

[Option -Oi](#)

C4301: Inline expansion done for function call

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler was replacing the function call with the code of the function to be called.

Example

```
inline int f(int i) {
    return i+1;
}
void main() {
    int i=f(3); // gets replaced by i=3+1;
}
```

Tips

To force the compiler to inline function calls use the keyword "inline".

See also

[Option -Oi](#)

C4302: Could not generate inline expansion for this function call

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler could not replace the function call with the code of the function to be called. The expression containing the function call may be too complex, the function to be called may be recursive or too complex.

Example

```
inline int f(int i) {
    if(i>10) return 0;
    return f(i+1);
}
void main() {
    int i=f(3); // Cannot inline,
                 // because f() contains a recursive call
}
```

Tips

To have the same effect as inlining the function, replace the call with the code of the function manually.

C4303: Illegal pragma <name>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An parsing error was found inside of the given pragma.

Example

```
#pragma INLINE Inline it!
/* pragma INLINE does not expect any arguments */
void foo(void) {
}
```

Tips

Check the exact definition of this pragma.

Use comments to add text behind a pragma.

C4400: Comment not closed

[FATAL]

Description

The preprocessor has found a comment which has not been closed.

Example

```
/*
```

Tips

Close the comment.

C4401: Recursive comments not allowed

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A recursive comment has been found (a comment with inside a comment).

Example

```
/* /* nested comment */
```

Tips

Either correct the comment, use '#if 0' - '#endif' or the C++ like comment '\\':

```
#if 0
    /* nested comment */
#endif
// /* /* nested comment */
```

C4402: Redefinition of existing macro '<MacroName>'

[FATAL]

Description

It is not allowed to redefine a macro.

Example

```
#define ABC 10
#define ABC 20
```

Tips

Correct the macro (e.g. using another name).

C4403: Macro-buffer overflow

[FATAL]

Description

There are more than 10'000 macros in a single compilation unit.

Example

```
#define macro0
#define macro1
...
#define macro10000
```

Tips

Simplify the compilation unit to reduce the amount of macro definitions.

See also

Limitations

C4404: Macro parents not closed

[FATAL]

Description

In a usage of a macro with parameters, the closing parenthesis is not present.

Example

```
#define cat(a,b) (a##b)
int i = cat(12,34;
```

Tips

Add a closing ')'.

C4405: Include-directive followed by illegal symbol

[FATAL]

Description

After an #include directive, there is an illegal symbol (not a file name in double quotes or within '<' and '>').

Example

```
#include <string.h> <<< message C4405 here
```

Tips

Correct the #include directive.

C4406: Closing '}' missing

[FATAL]

Description

There is a missing closing '}' for the include directive.

Example

```
#include <string.h
```

Tips

Correct the #include directive.

C4407: Illegal character in string or closing '}' missing

[FATAL]

Description

Either there is a non-printable character (as control characters) inside the file name for the #include directive or the file name is not enclosed with '<' and '>'.

Example

```
#include <abc.h[control character]
```

Tips

If there are non-printable characters inside the file name, remove them.

If there is a missing '>', add a '>' to the end of the file name.

C4408: Filename too long

[FATAL]

Description

A include file name longer than 2048 characters has been specified.

Example

```
#include <VeryLongFilename.....>
```

Tips

Shorten the file name, e.g. using a relative path or setting up the include file path in the default.env environment file.

See also

Limitations

C4409: a ## b: the concatenation of a and b is not a legal symbol

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The concatenation operator ## is used to concatenate symbols. If the resulting symbol is not a legal one, this message is issued.

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Example

```
#define concat(a,b) a ## b
void foo(int a) {
    a concat(=,@) 5; // message: =@ is not a legal symbol
}
```

Tips

Check your macro definition. Generate a preprocessor output (option [-Lp](#)) to find the problem.

C4410: Unbalanced Parentheses

[FATAL]

Description

The number of opening parentheses '(' and the number of closing parentheses ')' does not match.

Tips

Check your macro definition. Generate a preprocessor output (option [-Lp](#)) to find the problem.

C4411: Maximum number of arguments for macro expansion reached

[FATAL]

Description

The compiler has reached the limit for the number of macro arguments for a macro invocation.

Example

```
#define A0(p1,p2,...,p1024) (p1+p2+...+p1024)
#define A1(p1,p2,...,p1024) A0(p1+p2+...+p1024)

void foo(void) {
    A1(1,2,...,1024); // message C4411 here
}
```

Tips

Try to avoid such a huge number of macro parameters, use simpler macros instead.

See also

[Limitations](#)

C4412: Maximum macro expansion level reached

[FATAL]

Description

The compiler has reached the limit for recursive macro expansion. A recursive macro is if a macro depends on another macro. The compiler also stops macro expansion with this message if it seems to be an endless macro expansion.

Example

```
#define A0 0
#define A1 A0
#define A2 A1
...
```

Tips

Try to reduce huge dependency list of macros.

See also

[Limitations](#)

C4413: Assertion: pos failed

[FATAL]

Description

This is a compiler internal error message only. It happens if during macro expansion the macro definition position is not the same as during the initial macro scanning.

Tips

If you encounter this message, please send us a preprocessor output (option [-Lp](#)).

C4414: Argument of macro expected

[FATAL]

Description

The preprocessor tries to resolve a macro expansion. However, there is no macro argument given after the comma separating the different macro arguments.

Example

```
#define Macro(a,b)
void foo(void) {
    Macro(,);
}
```

Tips

Check your macro definition or usage. Generate a preprocessor output (option [-Lp](#)) to find the problem.

C4415: ')' expected

[FATAL]

Description

The preprocessor expects a closing parenthesis. This may happen if a preprocessor macro has been called more argument than previously declared.

Example

```
#define A(a,b)  (a+b)

void main(void) {
    int i = A(3,4,5); // message C4415 here
}
```

Tips

Use the same number of arguments as declared in the macro.

C4416: Comma expected

[FATAL]

Description

The preprocessor expects a comma at the given position.

Tips

Check your macro definition or usage.

C4417: Mismatch number of formal, number of actual parameters

[FATAL]

Description

A preprocessor macro has been called with a different number of argument than previously declared.

Example

```
#define A(a,b)  (a+b)
void main(void) {
    int i = A(3,5,7); // message C4417 here
}
```

Tips

Use the same number of arguments as declared in the macro.

C4418: Illegal escape sequence

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An illegal escape sequence occurred. A set of escape sequences is well defined in ANSI C. Additionally there are two forms of numerical escape sequences. The compiler has detected an escape sequence which is not covered by ANSI.

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Example

```
char c= '\p';
```

Tips

Remove the escape character if you just want the character.

When this message is ignored, the compiler issued just the character without considering the escape character. So '\p' gives just a 'p' when this message is ignored.

To specify an additional character use either the octal or hexadecimal form of numerical escape sequences.

```
char c_space1= ' '; /* conventional space */  
char c_space2= '\x20'; /* space with hex notation */  
char c_space3= '\040'; /* space with octal notation */
```

Only 3 digits are read for octal numbers, so when you specify 3 digits, then, there is no danger of combining the numerical escape sequence with the next character in sequence. Hexadecimal escape sequences should be terminated explicitly.

```
const char string1[] = " 0"; // gives " 0"  
const char string2[] = "\400"; // error because 0400 > 255  
const char string3[] = "\x200"; // error because 0x200 >  
                           255  
const char string4[] = "\0400"; // gives " 0"  
const char string5[] = "\x20" "0"; // gives " 0"
```

See also

[List of Escape Sequences](#)

C4419: Closing “ missing

[FATAL]

Description

There is a string which is not terminated by a double quote. This message is also issued if the not closed string is at the end of the compilation unit file.

C4420: Illegal character in string or closing " missing

[FATAL]

Description

A string contains either an illegal character or is not terminated by a closing double quote.

Example

```
#define String "abc
```

Tips

Check your strings for illegal characters or if they are terminated by a double quote.

C4421: String too long

[FATAL]

Description

Strings in the preprocessor are actually limited to 8192 characters.

Tips

Use a smaller string or try to split it up.

See also

Limitations

C4422: ' missing

[FATAL]

Description

To define a character, it has to be surrounded by single quotes (').

Example

```
#define CHAR 'a
```

Tips

Add a single quote at the end of the character constant.

C4423: Number too long

[FATAL]

Description

During preprocessing, the maximum length for a number has been reached.
Actually this length is limited to 8192 characters.

Example

```
#define MyNum 12345.....8193 // 8193 characters
```

Tips

Probably there is a typing error or the number is just too big.

See also

Limitations

C4424: # in substitution list must be followed by name of formal parameter

[FATAL]

Description

There is a problem with the '#' operator during preprocessing, because there is no legal name as formal parameter specified.

Example

```
#define cat(a,b) a #
void foo(void) {
    i = cat(3,3);
}
```

Tips

Check your macro definition or usage. Generate a preprocessor output (option [-Lp](#)) to find the problem.

C4425: ## in substitution list must be preceded and followed by a symbol

[FATAL]

Description

There is a problem with the string concatenation ‘##’ operator during preprocessing, because there is no legal name as formal parameter specified.

Example

```
#define cat(a,b) a ##
void foo(void) {
    i = cat(3,3);
}
```

Tips

Check your macro definition or usage. Generate a preprocessor output (option [-Lp](#)) to find the problem.

C4426: Macro must be a name

[FATAL]

Description

There has to be a legal C/C++ ident to be used as a macro name. An ident has to be start with a normal letter (e.g. ‘a’..’Z’) or any legal ident symbol.

Example

```
#define "abc"
```

Tips

Use a legal macro name, e.g. not a string or a digit.

C4427: Parameter name expected

[FATAL]

Description

The preprocessor expects a name for preprocessor macros with parameters.

Example

```
#define A(3) (B)
```

Tips

Do not use numbers or anything else than a name as macro parameter names.

C4428: Maximum macro arguments for declaration reached

[FATAL]

Description

The preprocessor has reached the maximum number of arguments allowed for a macro declaration.

Example

```
#define A(p0, p1, ..., p1023, p1024) (p0+p1+...p1024)
```

Tips

Try to split your macro into two smaller ones.

See also

Limitations

C4429: Macro name expected

[FATAL]

Description

The preprocessor expects macro name for the #undef directive.

Example

```
#undef #xyz
```

Tips

Use only a legal macro name for the #undef directive.

C4430: Include macro does not expand to string

[FATAL]

Description

The file name specified is not a legal string. A legal file name has to be surrounded with double quotes “”.

Example

```
#define file 3  
#include file
```

Tips

Specify a legal file name.

C4431: Include "filename" expected

[FATAL]

Description

There is no legal file name for a include directive specified. A file name has to be non-empty and surrounded either by ‘<‘ and ‘>’ or by double quotes “””.

Example

```
#include <>
```

Tips

Specify a legal file name.

C4432: Macro expects ‘(‘

[FATAL]

Description

While expanding macros, the preprocessor expects here an opening parenthesis to continue with macro expansion.

Tips

Check your macro definition or usage. Generate a preprocessor output (option [-Lp](#)) to find the problem.

C4433: Defined <name> expected

[FATAL]

Description

Using ‘defined’, it can be checked if a macro is defined or not. However, there has to be a name used as argument for defined.

Example

```
#if defined()  
#endif
```

Tips

Specify a name for the defined directive, e.g. #if defined(abc).

C4434: Closing ')' missing

[FATAL]

Description

During macro expansion, the preprocessor expects a closing parenthesis to continue.

Tips

Check your macro definition or usage. Generate a preprocessor output (option [-Lp](#)) to find the problem.

C4435: Illegal expression in conditional expression

[FATAL]

Description

There is an illegal conditional expression used in a #if or #elif directive.

Example

```
#if (3*)  
#endif
```

Tips

Check the conditional expression.

C4436: Name expected

[FATAL]

Description

The preprocessor expects a name for the #ifdef and #ifndef directive.

Example

```
#ifndef 3333_H // << C4436 here */  
#define 3333_H  
#endif
```

Tips

Check if a legal name is used, e.g. it is not legal to start a name with a digit.

C4437: Error-directive found: <message>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The preprocessor stops with this message if he encounters an #error directive.

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Example

```
#error "error directive"
```

Tips

Check why the preprocessor evaluates to this error directive. Maybe you have forgotten to define a macro which has caused this error directive.

C4438: Endif-directive missing

[FATAL]

Description

All #if or #ifdef directives need a #endif at the end. If the compiler does not find one, this message is issued.

Example

```
#if 1
```

Tips

Check where the #endif is missing. Generate a preprocessor output (option [-Lp](#)) to find the problem.

C4439: Source file <file> not found

[FATAL]

Description

The compiler did not find the source file to be used for preprocessing.

Tips

Check why the compiler was not able to open the indicated file. Maybe the file is not accessible any more or locked by another application.

C4440: Unknown directive: <directive>

[FATAL]

Description

The preprocessor has detected a directive which is unknown and which cannot be handled.

Example

```
#notadirective
```

Tips

Check the directive. Maybe it is a non-ANSI directive supported by another compiler.

C4441: Preprocessor output file <file> could not be opened

[FATAL]

Description

The compiler was not able to open the preprocessor output file. The preprocessor file is generated if the option [-Lp](#) is specified.

Tips

Check your macro definition or usage. Check the option [-Lp](#): the format specified may be illegal.

C4442: Endif-directive missing

[FATAL]

Description

The #asm directive needs a #endasm at the end. If the compiler does not find one, this message is issued.

Example

```
#asm
```

Tips

Check where the #endasm is missing. Generate a preprocessor output (option [-Lp](#)) to find the problem.

C4443: Undefined Macro 'MacroName' is taken as 0

[DISABLE, INFORMATION, WARNING, ERROR]

Description

Whenever the preprocessor evaluates a condition and finds a identifier which was not defined before, he implicitly takes its value to be the integral value 0. This C style behavior may arise in hard to find bug. So when the header file, which actually defines the value is not included or when the macro name was entered incorrectly, for example with a different case, then the preprocessor "#if" and "#elif" instructions wont behave as expected.

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Example

```
#define _Debug_Me 1
...
void main(int i) {
    #if Debug_Me /* missing _ in front of _Debug_Me. */
        assert(i!=0);
```

```
#endif
}
```

The assert will never be encoded.

Tips

This warning is a good candidate to be mapped to an error. To save test macros defined in some header files, also test if the macros are defined:

```
#define _Debug_Me
...
void main(int i) {
#ifndef Debug_Me
#error /* macro must be defined above */
#endif
#if Debug_Me /* missing _ in front of _Debug_Me. */
    assert(i!=0);
#endif
}
```

The checking of macros with "#ifdef" and "#ifndef" cannot detect if the header file, a macro should define is really included or not. Note that using a undefined macro in C source will treat the macro as C identifier and so usually be remarked by the compiler. Also note a undefined macro has the value 0 inside of preprocessor conditions, while a defined macro with nothing as second argument of a "#define" replaces to nothing. E.g.

```
#define DEFINED_MACRO
#ifndef UNDEFINED_MACRO // evaluates to 0, giving this
warning
#endif
#if DEFINED_MACRO // error FATAL: C4435: Illegal
// expression in conditional expression
#endif
```

C4444: Line number for #line directive must be > 0 and <= 32767

[DISABLE, INFORMATION, WARNING, ERROR]

Description

ANSI requires that the line number for the line directive is greater zero or smaller-equal than 32767. If this message occurs and it is currently not mapped to an error, the compiler sets the line number to 1.

Example

```
#line 0  
#line 32768 "myFile.c"
```

Tips

Specify a line number greater zero and smaller 32768.

For automatically generated code, which has such illegal line directives, you can move this error to a warning.

C4445: Line number for #line directive expected

[DISABLE, INFORMATION, WARNING, ERROR]

Description

ANSI requires that after the line directive a number has to follow.

Example

```
#line // << ERROR C4445  
#line "myFile.c" // << ERROR C4445
```

Tips

Specify a line number greater zero and smaller 32768.

C4446: Missing macro argument(s)

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In a macro 'call', one or more arguments are missing. The pre-processor replaces the parameter string with nothing.

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Example

```
#define f(A, B) int A B
void main(void){
    f(i,); // this statement will be replaced with 'int i;'
            // by the pre-processor.
}
```

Tips

Be careful with empty macro arguments, because the behavior is undefined in ANSI-C. So avoid it if possible.

C4447: Unexpected tokens following preprocessor directive - expected a newline

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler has detected that after a directive there was something unexpected. Directives are normally line oriented, thus the unexpected tokens are just ignored.

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Example

```
#include "myheader.h" something
```

Tips

Remove the unexpected tokens.

C4448: Warning-directive found: <message>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The preprocessor stops with this message if he encounters an #warning directive.

Note that this directive is only support if [-Ansi](#) is not set.

Note: The [pragma MESSAGE](#) does not apply to this message because it is issued in the preprocessing phase.

Example

```
#warning "warning directive"
```

Tips

Check why the preprocessor evaluates to this warning directive. Maybe you have forgotten to define a macro which has caused this directive.

C4449: Exceeded preprocessor #if level of 4092

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The preprocessor does by default only allow 4092 concurrently open #if directives. If more do happen, this message is printed.

Usually this message does only happen because of a programming error.

Example

```
#if 1 // 0
#if 2 // 0
#if 3 // 0
.....
#if 4092 // 0
```

Tips

Check why there are that many open preprocessor if's.

Are the #endif's missing?

C4700: Illegal pragma TEST_ERROR

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The pragma TEST_ERROR is for internal use only. It is used to test the message management and also to test error cases in the compiler.

C4701: pragma TEST_ERROR: Message <ErrorNumber> did not occur

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The pragma TEST_ERROR is for internal use only. It is used to test the message management and also to test error cases in the compiler.

C4800: Implicit cast in assignment

[DISABLE, INFORMATION, WARNING, ERROR]

Description

An assignment requiring an implicit cast was made.

Tips

Check, if the casting results in correct behavior.

C4801: Too many initializers

[ERROR]

Description

The braced initialization has too many members.

Example

```
char name[ 4 ]={ 'n' , 'a' , 'm' , 'e' , 0 } ;
```

Tips

Write the correct number of members in the braced initializer.

C4802: String-initializer too large

[ERROR]

Description

The string initializer was too large.

Tips

Take a shorter string, or try to allocate memory for your string in an initialization function of the compilation unit.

C4900: Function differs in return type only

[ERROR] , C++

Description

The overloaded function has the same parameters, but not the same return type.

Example

```
void f(int);  
void f();  
int f(int); // error
```

Tips

A function redeclaration with the same parameters must have the same return type than the first declaration.

C5000: Following condition fails: `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)`

[ERROR]

Description

Type sizes has been set to illegal sizes. For compliance with the ANSI-C rules, the type sizes of char, short, int, long and long long has to in a increasing order, e.g. setting char to a size of two and int to a size of one will violate this ANSI rule.

Example

-Tc2i1

Character as two bytes and int as one byte is illegal.

Tips

Change the [-T](#) option.

C5001: Following condition fails: `sizeof(float) <= sizeof(double) <= sizeof(long double) <= sizeof(long long double)`

[ERROR]

Description

Your settings of the Standard Types are wrong!

Example

-Tf4d2

Float as IEEE64 and double as IEEE32 is illegal.

Tips

Set your Standard Types correctly!

See also

[Change the -T option.](#)

C5002: Illegal type

[ERROR]

Description

A unknown or illegal type occurred.

This error may happen as consequence of another error creating the illegal type.

Tips

Check for other errors happening before.

C5003: Unknown array-size

[ERROR]

Description

A compiler internal error happened!

Tips

Please contact your distributor.

C5004: Unknown struct-union-size

[ERROR]

Description

A compiler internal error happened!

Tips

Please contact your distributor.

C5005: PACE illegal type

[ERROR]

Description

A compiler internal error happened!

Tips

Please contact your distributor.

C5006: Illegal type settings for HIWARE Object File Format

[ERROR]

Description

For HIWARE object file format (option [-Fh](#), [-F7](#) and default if no other object file format is selected) the char type must always be of size 1. This limitation is because there may not be any new types introduced in this format and 1 byte

types are used internally in the compiler even if the user only need multibyte characters.

For the strict HIWARE object file format (option [-F7](#)) the additional limitation that the enum type has the size 2 bytes, and must be signed, is checked with this message.

The HIWARE Object File Format (-Fh) has following limitations:

- The type char is limited to a size of 1 byte
- Symbolic debugging for enumerations is limited to 16bit signed enumerations
- No symbolic debugging for enumerations
- No zero bytes in strings allowed (zero byte marks the end of the string)
- The strict HIWARE V2.7 Object File Format (option [-F7](#)) has some limitations:
 - The type char is limited to a size of 1 byte
 - Enumerations are limited to a size of 2 and has to be signed
 - No symbolic debugging for enumerations
 - The standard type ‘short’ is encoded as ‘int’ in the object file format
 - No zero bytes in strings allowed (zero byte marks the end of the string)

Example

COMPOPTIONS= [-Te2](#) [-Fh](#)

Tips

Use [-Fh](#) HIWARE object file format to change the enum type. To change the char type, only the ELF object file format can be used (if supported).

Note that not all backends allow the change of all types.

See also

[Option for object file format -Fh/-F7/-F1/-F2](#)

Option for Type Setting -T

C5100: Code size too large

[ERROR]

Description

The code size is too large for this compiler.

Example

```
// a large source file
```

Tips

Split up your source into several compilation units!

See also

Limitations

C5200: 'FileName' file not found

[ERROR]

Description

The specified source file was not found.

Example

```
#include "notexisting.h"
```

Tips

Specify the correct path and name of your source file!

See also

Input Files

C5250: Error in type settings: <Msg>

[ERROR]

Description

There is an inconsistent state in the type option settings. E.g. it is illegal to have the ‘char’ size larger than the size for the type ‘short’.

Tips

Check the -T option in your configuration files. Check if the option is valid.

See also

Option -T

C5300: Limitation: code size '<actualSize>' > '<limitSize>' bytes

[ERROR]

Description

You have a limited version of the compiler or reached the limitation specified in the license file. The actual demo limitation is 1024 bytes of code for 8/16bit targets and 3KByte for 32bit targets (without a license file). Depending on the license configuration, the code size limit may be specified in the license file too.

Tips

Check if you have enough licenses if you are using a floating license configuration. Check for the correct location of the license file. Get a license for a full version of the compiler, or for a code size upgrade.

C5302: Couldn't open the object file '<FileName>'

[FATAL]

Description

The compiler cannot open the object file for writing.

Tips

Check if there is already an object file with the same name but used by another application. Check if the object file is marked as read-only or locked by another application.

C5320: Cannot open logfile '<FileName>'

[FATAL]

Description

The compiler cannot open the logfile file for writing.

Tips

Check if there is already a file with the same name but used by another application. Check if the file is marked as read-only or locked by another application.

See also

Option -Ll

C5350: Wrong or invalid encrypted file '<File>' (<MagicValue>)

[FATAL]

Description

The compiler cannot read an encrypted file because the encrypted file magic value is wrong.

Tips

Check if the file is a valid encrypted file.

See also

Option -Eencrypt

Option -Ekey

C5351: Wrong encryption file version: '<File>' (<Version>)

[FATAL]

Description

The compiler cannot read the encrypted file because the encryption version does not match.

Tips

Check if you have a valid license for the given encryption version. Check if you use the same license configuration for encryption and encrypted file usage.

See also

Option -Eencrypt

Option -Ekey

C5352: Cannot build encryption destination file: '<FileSpec>'

[FATAL]

Description

Building the encryption destination file name using the <FileSpec> was not possible.

Tips

Check your FileSpec if it is legal.

See also

Option -Eencrypt

Option -Ekey

C5353: Cannot open encryption source file: '<File>'

[FATAL]

Description

It was not possible to open the encryption source file.

Tips

Check if the source file exists.

See also

Option -Eencrypt

Option -Ekey

C5354: Cannot open encryption destination file: '<File>'

[FATAL]

Description

The compiler was not able to write to the encryption destination file.

Tips

Check if you have read/write access to the destination file. Check if the destination file name is a valid one. Check if the destination file is locked by another application.

See also

Option -Eencrypt

Option -Ekey

C5355: Encryption source '<SrcFile>' and destination file '<DstFile>' are the same

[FATAL]

Description

The encryption source file and the destination file are the same. Because it is not possible to overwrite the source file with the destination file, encryption is aborted.

Tips

Change the encryption destination file name specification.

See also

Option -Eencrypt

Option -Ekey

C5356: No valid license for encryption support

[FATAL]

Description

It was not possible to check out the license for encryption support.

Tips

Check your license configuration. Check if you have a valid encryption license.

See also

Option -Lic

C5650: Too many locations, try to split up function

[FATAL]

Description

The internal data structure of the compiler blows up. One of your functions is too large.

Tips

Split up the function that causes the message.

See also

Limitations

C5651: Local variable <variable> may be not initialized

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler issues this warning if no dominant definition is found for a local variable when it is referenced. Ignore this message if the local variable is defined implicitly (e.g. by inline assembly statements).

Example

```
int f(int param_i) {
    int local_i;

    if(param_i == 0) {
        local_i = 1;      /* this definition for local_i does
                           */
                           /* NOT dominate following usage */
    }
    return local_i;     /* local_i referenced: no dominant */
                       /*definition for local_i found */
}
```

Tips

Review the code and initialize local variables at their declaration (e.g. local_i = 0).

C5660: Removed dead code

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler has optimized some unused code away.

Tips

Sometimes these compiler message shows some problem in the C code.

C5661: Not all control paths return a value

[DISABLE, INFORMATION, WARNING, ERROR]

Description

There are one or more control paths without a return <value> statement. Your function does not return a value in every case.

Example

```
int glob;  
int test(void) {  
    if (glob) {  
        return 1;  
    }  
}
```

Tips

Do return a value in all control flows.

C5662: Generic Inline Assembler Error <string>

[DISABLE, INFORMATION, WARNING, ERROR], SICG

Description

An error occurred during intermediate code generation for inline assembly instructions. The particular error is described in textual form.

Example

The inline assembly code taking the address of a local variable, which is not allocated on the stack.

C5680: HLI Error <Detail>

[ERROR]

Description

This is a generic error message issued by the HLI.

C5681: Unexpected Token

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The HLI issues this message for unexpected tokens during parsing.

C5682: Unknown fixup type

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The HLI issues this message if the parser finds an unknown fixup type.

Tips

If this error occurs, check the list of available fixup types and make sure there are no spelling errors.

C5683: Illegal directive <Ident> ignored

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The HLI issues this message for directives it knows of, but which are ignored.

C5684: Instruction or label expected

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The HLI issues this message if the HLI parser expects an instruction (or directive) or a label, but finds something else. This error shows up if the assembly line is syntactically wrong, or if the assembly instruction or directive is not known.

C5685: Comment or end of line expected

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The HLI issues this message if it finds unexpected tokens after the operands of an assembly instructions. This may be due to an error in the operand syntax or because the comment is not properly delimited.

C5686: Undefined label <Ident>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The inline assembler issues this message if an undeclared ident is used but not defined as a label. Note that a spelling error in the name of a variable or register may result in this error as well.

Example

In the following code, the label `loop` is not defined

```
void example (void){  
    asm {  
        bne    loop  
    }  
}
```

C5687: Label <Ident> defined more than once

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The HLI issues this message for labels which are defined more than once within the same function.

C5688: Illegal operand

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The HLI issues this message if an operand is syntactically correct, but it is out of range, or is not constant, the fixup type is not appropriate, etc.

Tips

This error message is relatively unspecific about the particular error. If the reason for this error is unclear, we suggest that you write the same instruction with simpler operands to find out what caused the error.

C5689: Constant or object expected

[ERROR]

Description

The HLI issues this message if the HLI parser expects a constant or an object, but finds another token.

C5690: Illegal fixup for constant

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The HLI issues this message if a fixup for a constant is not possible or the constant is out of range.

C5691: Instruction operand mismatch

[[ERROR](#)]

Description

The HLI issues this message if an assembly instruction is not available. Note that the assembly instruction is still known and may be available for other operands.

Tips

This error message is relatively unspecific. It shows up if the type of operands does not match (e.g., a register instead of an immediate) or the number of operands is different, or the overall operand syntax is wrong. If the error is not obvious, we suggest that you find the error by comparison with a corresponding instruction, but with plain assembly operands.

C5692: Illegal fixup

[[DISABLE](#), [INFORMATION](#), [WARNING](#), [ERROR](#)]

Description

The HLI issues this message if a particular fixup is illegal the way it is used.

C5693: Cannot generate code for <Error>

[[DISABLE](#), [INFORMATION](#), [WARNING](#), [ERROR](#)]

Description

The compiler can not generate code because of the specified reason.

Possible reasons are:

- code for intrinsic function cannot be generated because of wrong arguments or other preconditions

- the compiler does not support a certain construct. Note: this is a generic message.

C5700: Internal Error <ErrorNumber> in '<Module>', please report to <Producer>

[FATAL]

Description

The Compiler is in an internal error state. Please report this type of error as described in the chapter [Bug Report](#)

This message is used while the compiler is not investigating a specific function.

Example

no example known.

Tips

Sometimes these compiler bugs occur in wrong C Code. So look in your code for incorrect statements.

Simplify the code as long as the bug exists. With a simpler example, it is often clear what is going wrong and how to avoid this situation.

Try to avoid compiler optimization by using the volatile keyword.

Please report errors for which you do have a work around.

See also

Chapter [Bug Report](#)

C5701: Internal Error #<ErrorNumber> in '<Module>' while compiling file '<File>', procedure '<Function>', please report to <Producer>

[FATAL]

Description

The Compiler is in an internal error state. Please report this type of error as described in the chapter [Bug Report](#).

This message is used while the compiler is investigating a specific function.

Example

no example known.

Tips

Sometimes these compiler bugs occur in wrong C Code. So look in your code for incorrect statements.

Simplify the code as long as the bug exists. With a simpler example, it is often clear what is going wrong and how to avoid this situation.

Try to avoid compiler optimization by using the volatile keyword.

Please report errors for which you do have a work around.

See also

Chapter [Bug Report](#)

C5702: Local variable ‘<Variable>’ declared in function ‘<Function>’ but not referenced

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected a local variable which is not used.

Example

```
void main(void) {  
    int i;  
}
```

Tips

Remove the variable if it is never used. If it is used in some situations with conditional compilation, use the same conditions in the declaration as in the usages.

C5703: Parameter ‘<Parameter>’ declared in function ‘<Function>’ but not referenced

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected a named parameter which is not used.

In C parameters in function definitions must have names. In C++ parameters may have a name. If it has no name, this warning is not issued.

This warning may occur in cases where the interface of a function is given, but not all parameters of all functions are really used.

Example

```
void main(int i) {  
}
```

Tips

If you are using C++, remove the name in the parameter list. If you are using C, use a name which makes clear that this parameter is intentionally not used as, for example “dummy”.

C5800: User requested stop

[DISABLE, INFORMATION, WARNING, ERROR]

Description

This message is used when the user presses the stop button in the graphical user interface.

Also when the compiler is closed during a compilation, this message is issued.

Tips

By moving this message to a warning or less, the stop functionality can be disabled.

C5900: Result is zero

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected operation which results in zero and is optimized.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
i = j-j; // optimized to i = 0;
```

Tips

If it is a programming error, correct the statement.

C5901: Result is one

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected an operation which results in one. This operation is optimized.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
i = j/j; // optimized to i = 1;
```

Tips

If it is a programming error, correct the statement.

C5902: Shift count is zero

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected an operation which results in a shift count of zero.
The operation is optimized.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
i = j<<(j-j); // optimized to i = j;
```

Tips

If it is a programming error, correct the statement.

C5903: Zero modulus

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected a % operation with zero. Because the modulus operation implies also a division (division by zero), the compiler issues a warning.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
i = j%0; // error
```

Tips

Correct the statement.

C5904: Division by one

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected a division by one which is optimized.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
i = j/1; // optimized to i = j;
```

Tips

If it is a programming error, correct the statement.

C5905: Multiplication with one

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected a multiplication with one which is optimized.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
i = j*1; // optimized to i = j;
```

Tips

If it is a programming error, correct the statement.

C5906: Subtraction with zero

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected a subtraction with zero which is optimized.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
i = j-(j-j); // optimized to i = j;
```

Tips

If it is a programming error, correct the statement.

C5907: Addition replaced with shift

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected a addition with same left and right expression which is optimized and replaced with a shift operation.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
i = j+j; // optimized to i = j<<1;
```

Tips

If it is a programming error, correct the statement.

C5908: Constant switch expression

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected a constant switch expression. The compiler optimizes and reduces such a switch expression

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
switch(2){  
    case 1: break;  
    case 2: i = 0; break;  
    case 3: i = 7; break;  
}; // optimized to i = 0;
```

Tips

If it is a programming error, correct the statement.

C5909: Assignment in condition

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected an assignment in a condition. Such an assignment may result from a missing '=' which is normally a programming error.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
if (i = 0) // should be 'i == 0';
```

Tips

If it is a programming error, correct the statement.

C5910: Label removed

[DISABLE, [INFORMATION](#), WARNING, ERROR]

Description

The Compiler has detected a label which can be optimized .

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
switch(i) {
    Label: i = 0; // Labeled removed
    ...
    if (cond) {
        L2: // L2 not referenced: Label removed
        ...
    } else {
    }
```

Tips

Do not use normal labels in switch statements. If it is a switch case label, do not forget to add the ‘case’ keyword.

C5911: Division by zero at runtime

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected zero division. This is not necessarily an error (see below).

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
void RaiseDivByZero(void) {
    int i = i/0; // Division by zero!
}
```

Tips

Maybe the zero value the divisor results from other compiler optimizations or because a macro evaluates to zero. It is a good idea to map this warning to an error (see Option [-WmsgSe](#)).

C5912: Code in 'if' and 'else' part are the same

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected that the code in the ‘if’ and the code in the ‘else’ part of an ‘if-else’ construct is the same. Because regardless of the condition in the ‘if’ part, the executed code is the same, so the compiler replaces the condition with ‘TRUE’ if the condition does not have any side effects. There is always a couple of this message, one for the ‘if’ part and one for the ‘else’ part.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
if (condition) { // replaced with 'if (1) {'  
    statements; // message C5912 here ...  
} else {  
    statements; // ... and here  
}
```

Tips

Check your code why both parts are the same. Maybe different macros are used in both parts which evaluates to the same values.

C5913: Conditions of 'if' and 'else if' are the same

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected that the condition in an ‘if’ and a following ‘else if’ expression are the same. If the first condition is true, the second one is never evaluated. If the first one is FALSE, the second one is useless, so the compiler replaces the second condition with ‘FALSE’ if the condition does not have any side effects. There is always a couple of this message, one for the ‘if’ part and one for the ‘if else’ part.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
if (condition) { // message C5913 here  
    ...;  
} else if(condition) { // here, condition replaced with 0  
    ...  
}
```

Tips

Check your code why both conditions are the same. Maybe different macros are used in both conditions which evaluates to the same values.

C5914: Conditions of 'if' and 'else if' are inverted

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected that the condition in an ‘if’ and a following ‘else if’ expression are just inverted. If the first condition is true, the second one is never evaluated (FALSE). If the first one is FALSE, the second one is TRUE, so the compiler replaces the second condition with ‘TRUE’ if the condition does not have any side effects. There is always a couple of this message, one for the ‘if’ condition and one for the ‘if else’ condition.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
if (condition) { // message C5914 here ...
    ...
} else if(!condition) { // here, condition replaced with 1
    ...
}
```

Tips

Check your code why both conditions are inverted. Maybe different macros are used in both parts which evaluates to the same values.

C5915: Nested 'if' with same conditions

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected that the condition in an ‘if’ and a nested ‘if’ expression have the same condition. If the first condition is true, the second one is always true. If the first one is FALSE, the second one is FALSE too, so the compiler replaces the second condition with ‘TRUE’ if the condition does not

have any side effects. There is always a couple of this message, one for the first ‘if’ condition and one for nested ‘if’ condition.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
if (condition) { // message C5915 here ...
    if(!condition) { // here, condition replaced with 1
        ...
    }
```

Tips

Check your code why both conditions are the same. Maybe different macros are used in both parts which evaluates to the same values.

C5916: Nested ‘if’ with inverse conditions

[DISABLE, [INFORMATION](#), WARNING, ERROR]

Description

The Compiler has detected that the condition in an ‘if’ and a nested ‘if’ expression have the inverse condition. If the first condition is true, the second one is always false. If the first one is FALSE, the second one is TRUE, so the compiler replaces the second condition with ‘FALSE’ if the condition does not have any side effects. There is always a couple of this message, one for the first ‘if’ condition and one for nested ‘if’ condition.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
if (condition) { // message C5916 here ...
    if(!condition) { // here, condition replaced with 0
        ...
    }
```

Tips

Check your code why both conditions are the same. Maybe different macros are used in both parts which evaluates to the same values.

C5917: Removed dead assignment

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected that there is an assignment to a (local) variable which is not used afterwards.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
int a;  
...  
a = 3 // message C5917 here ...  
} /* end of function */
```

Tips

If you want to avoid this optimization, you can declare the variable as volatile.

C5918: Removed dead goto

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Compiler has detected that there is goto jumping to a just following label.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
goto Label; // message C5918 here ...  
Label:  
...
```

Tips

If you want to avoid this optimization, you can declare the variable as volatile.

C5919: Conversion of floating to unsigned integral

[DISABLE, INFORMATION, WARNING, ERROR]

Description

In ANSI-C the result of a conversion operation of a (signed) floating type to a unsigned integral type is undefined. One implementation may return 0, another the maximum value of the unsigned integral type or even something else. Because such behavior may cause porting problems to other compilers, a warning message is issued for this.

Example

```
float f = -2.0;  
unsigned long uL = f; // message C5919 here
```

Tips

To avoid the undefined behavior, first assign/cast the floating type to a signed integral type and then to a unsigned integral type.

See also

ISO/IEC 9899:1990 (E), page 35, chapter 6.2.1.3 Floating and integral:

"When a value of floating type is converted to integral type, the fractional part is discarded. the value of the integral part cannot be represented by the integral type, the behavior is undefined."

C5920: Inlining library function <function>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler has inlined or replaced the indicated library function. Inlining is done with the option [Option -OiLib](#). If you have your own implementation of the indicated library function, maybe you have to disable inlining for this function, because the semantic may be different.

See also

[Option -OiLib](#)

C5921: Shift count out of range

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler has detected that there is a shift count exceeding the object size of the object to be shifted. This is normally not a problem, but can be optimized by the compiler. For right shifts (>>), the compiler will replace the shift count with (sizeOfObjectInBits-1), that is a shift of a 16bit object (e.g. a short value) with a right shift by twenty is replaced with a right shift by 15.

This message may be generated during tree optimizations (Option [-Ont](#) to switch it off).

Example

```
unsigned char uch, res;  
res = uch >> 9; // uch only has 8 bits, warning here  
// will be optimized to 'res = uch>>7'
```

See also

[Option -Ont](#)

C6000: Creating Asm Include File <file>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

A new file was created containing assembler directives. This file can be included into any assembler file to automatically get information from C header files.

Tips

Use this feature when you have both assembler and C code in your project.

See also

[Option -La](#)

[Create Assembler Include Files](#)

[pragma CREATE_ASM_LISTING](#)

C6001: Could not Open Asm Include File because of <reason>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The Assembler Include file could not be opened. As reason either occurs the file name, which was tried to open or another description of the problem.

Tips

Try to specify the name directly.

Check the usage of the file name modifiers.

Check if the file exists and is locked by another application

See also

[Option -La](#)

[Create Assembler Include Files](#)

[pragma CREATE_ASM_LISTING](#)

C6002: Illegal pragma CREATE_ASM_LISTING because of <reason>

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The pragma CREATE_ASM_LISTING was used in a ill formed way.

Tips

After the pragma, the may only be a ON or OFF.

ON and OFF are case sensitive and must not be surrounded by double quotes.

For example:

#pragma CREATE_ASM_LISTING ON

or

#pragma CREATE_ASM_LISTING OFF

See also

[Option -La](#)

[Create Assembler Include Files](#)

[pragma CREATE_ASM_LISTING](#)

Messages of XGATE Back End

This section contains message descriptions specific for the XGATE compiler.

All messages specific for the XGATE have a number of the form C22XXX, where XXX is a different number for every message.

Messages

Messages of XGATE Back End

C22000: XGATE stack pointer is not balanced

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The XGATE stack pointer register does not contain the same value for all control paths. If this message happens for C code, then it could indicate a compiler error. However this message can legally occur if HLI (High Level Inline Assembly) is used.

Example

```
int _carry(void);
void test(void) {
    int save_r1= _carry();
    if (save_r1) {
        __asm LDW R6, (R0, R7+);
    }
}
```

C22001: XGATE stack frame changed by inline assembler

[DISABLE, INFORMATION, WARNING, ERROR]

Description

The compiler detected that the stack is not correctly managed in HLI code.

Example:

```
int _carry(void);
void test(void) {
    int save_r1= _carry();
    if (save_r1) {
        __asm STW R1, (R0, -R7);
    }
}
```

C22002: Wrong type or number of formal parameters in interrupt handler

[ERROR]

Description

For the XGATE, interrupt functions do automatically get the R1 parameter set with the second entry at every vector table. This feature is supported by the compiler with a single 8 or 16 bit parameter. Interrupt functions with more or larger parameters than this do issue the C22002 message.

Example

```
struct Params {  
    int p0;  
    int p1;  
};  
void interrupt test(struct Params p) {  
}
```

Tips

Use a restricted pointer to pass the address of multiple parameters.

```
struct Params {  
    int p0;  
    int p1;  
};  
void interrupt test(struct Params* restrict p) {  
}
```

Messages

Messages of XGATE Back End

ANSI Library Reference

Library Files

Directory Structure

The library files are delivered in the following structure:

```
<install>\lib\<target>c\      /* readme files, make files */  
<install>\lib\<target>c\src /* library source files (C & C++)*/  
<install>\lib\<target>c\include /* library include files */  
<install>\lib\<target>c\lib     /* default library files */  
<install>\lib\<target>c\prm     /* Linker parameter files */
```

Check out the `readme.txt` located in the library folder with additional information on memory models and library file names.

How to Generate Library

In the directory structure above, a CodeWarrior `.mcp` file is provided to build all the libraries and the startup code object files. Simply load the `.mcp` into CodeWarrior and build all the targets.

Common Source Files

[Table 3.1](#) lists the source and header files of the Standard ANSI Library that are not target dependent.

Table 3.1 Standard ANSI Library—Target Independent Source and Header Files

Source File	Header File
alloc.c	
assert.c	assert.h
ctype.c	ctype.h
	errno.h
heap.c	heap.h
	limits.h
math.c, mathf.c	math.h , ieemath.h, float.h
printf.c, scanf.c	stdio.h
signal.c	signal.h
	stdarg.h
	stddef.h
stdlib.c	stdlib.h
string.c	string.h
	time.h

Startup Files

Because every memory model needs special startup initialization, there are also startup object file compiled with different Compiler option settings (see Compiler options for details).

The correct startup file has to be linked with your application depending on the memory model chosen or depending if your application contains C++ modules with global constructors/destructors or not. The floating point format used does not matter for the startup code.

Note that the library files contain a generic startup written in C as example doing all needed tasks for a startup:

- Zero Out

- Copy Down
- Register initialization
- C++ Constructor calls
- C++ Destructor calls
- Handling ROM libraries

Because not all of the above tasks may be needed for an application and for efficiency reasons, special startup is provided as well (e.g. written in HLI). But the version written in C can be used as well as example: just compile the ‘`startup.c`’ file with your memory/options settings and link it to your application.

Library Files

Most of the object files of the ANSI library are delivered in the form of a object library (see below).

Several Library files are bundled with the Compiler. The reason for different library files are different memory models or floating point formats.

The library files contain all necessary runtime functions used by the compiler and the ANSI Standard Library as well. The list files (extension `.lst`) contains a summary of all objects in the library file.

To link against a modified file which also exists in the library, it must be specified first in the link order.

Please check out the `readme.txt` located in the library structure
(`lib\<target>c\readme.txt`) for a list of all delivered library files and memory model/ options used.

Special Features

Not everything defined in the ANSI standard library makes sense in embedded systems programming. Therefore, not all functions have been implemented, and some have been left open to be implemented since they strongly depend on the actual setup of the target system.

This chapter describes and explains these points.

NOTE	All functions not implemented do a HALT when called. All functions are re-entrant, except rand and srand since these use a global
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------

variable to store the seed, which might give problems with light-weight processes. Another function using a global variable is [strtok](#), since it has been defined that way in the ANSI standard.

Memory Management - malloc, free, calloc, realloc; alloc.c, heap.c

File 'alloc.c' provides a full implementation of these functions, the only problem remaining is the question of where to put the heap, how big should it be and what should happen when we run out of heap memory.

All these points can be solved in file "heap.c". The heap simply is viewed as a large array, and there's a default error handling function. Feel free to modify this function or the size of the heap to suit your needs. The size of the heap is defined in libdefs.h, LIBDEF_HEAPSIZE.

Signals - signal.c

Signals have been implemented in a very rudimentary way - as traps. This means, function [signal](#) allows you to set a vector to some function of your own (which of course should be a TRAP_PROC), while function [raise](#) is not implemented. If you decide to ignore a certain signal, a default handler doing nothing is installed.

Multi-byte Characters - mblen, mbtowc, wctomb, mbstowcs, wcstombs; stdlib.c

Since the compiler does not support multi-byte characters, all routines in "stdlib.c" dealing with those have not been implemented. If you need them, you'll have to write your own functions.

Program Termination - abort, exit, atexit; stdlib.c

Since programs in embedded systems usually are not expected to terminate, we only provide a minimum implementation of the first two functions, while [atexit](#) is not implemented at all. Both [abort](#) and [exit](#) simply perform a HALT.

I/O - printf.c

The [printf](#) library function is in the current version of the library sets not implemented in the ANSI libraries, but in the file "terminal.c".

This difference has been done because often no terminal is available at all other terminal depends highly on the user hardware.

The ANSI library contains several functions which makes it simple to implement the [printf](#) function with all its special cases in a few lines.

The first, ANSI conform way is to allocate a buffer and then using the ANSI function [vsprintf](#).

Example:

```
int printf(const char *format, ...) {
    char outbuf[MAXLINE];
    int i;
    va_list args;
    va\_start(args, format);
    i = vsprintf(outbuf, format, args);
    va\_end(args);
    WriteString(outbuf);
    return i;
}
```

The value of MAXLINE defines the maximum size of any value of printf. The function `WriteString` is assumed to write one string to a terminal. There are several disadvantages of this solution:

- A buffer is needed which alone may use a large amount of RAM.
- Unimportant how large the buffer (MAXLINE) is, it is always possible that a buffer overflow occurs. Therefore this solution is not safe.

To avoid both disadvantages, we provide two non ANSI functions [vprintf](#) and [set_printf](#) in its newer library versions.

Because these functions are a non ANSI extension, they are not contained in the "[stdio.h](#)" header file.

Therefore its prototype must be specified before they are used:

```
int vprintf(const char *pformat, va_list args);
void set_printf(void (*f)(char));
```

The `set_printf` function installs a call back function, which is later called for every character which should be printed by [vprintf](#).

Remark that the standard ANSI C printf derivatives functions [sprintf](#) and [vsprintf](#) are also implemented by calls to `set_printf` and [vprintf](#). This way much of the code for all printf derivatives can be shared across them.

There is also a limitation of the current implementation of [printf](#). Because the callback function is not passed as argument to [vprintf](#), but hold in a global variable, all the [printf](#) derivatives are not reentrant. Even calls to different derivatives at the same time are not allowed.

A simple implementation of a [printf](#) with [vprintf](#) and `set_printf` is for example:

```
int printf(const char *format, ...){
    int i;
    va_list args;

    set_printf(PutChar);
    va_start(args, format);
    i = vprintf(format, args);
    va_end(args);
    return i;
}
```

The function `PutChar` is assumed to print one character to the terminal.

Another remark has to be made about the [printf](#) and [scanf](#) functions. We provides the full source code of all printf derivatives in "printf.c" and of scanf in "scanf.c". Usually many of the features of [printf](#) and [scanf](#) are not used by a specific application. The source code of the library modules printf and scanf contains switches (defines) to allow the use to switch off not used parts of the code. This especially includes the large floating point parts of [vprintf](#) and [vscanf](#).

Locales - locale.*

Has not been implemented.

ctype

ctype contains two sets of implementations for all functions. The standard is a set of macros which translate into accesses to a lookup table.

Since this table uses 257 bytes of memory, we also provide an implementation using real functions. These are accessible if you undefine the macros first (after "#undef isupper", [isupper](#) is translated into a call to function "isupper". Without the "undef", "isupper" is replaced by the corresponding macro.

Using the functions instead of the macros of course saves RAM and code size - at the expense of some additional function call overhead.

String conversions - `strtol`, `strtoul`, `strtod`, `stdlib.c`

To follow the ANSI requirements for string conversions, range checking has to be done. The variable “errno” is set accordingly and special limit values are returned. The macro “ENABLE_OVERFLOW_CHECK” is set to 1 by default. To reduce code size it’s recommended to switch off this macro (set ENABLE_OVERFLOW_CHECK to 0).

Library Structure

In this section, we will examine the various parts of the ANSI-C standard library, grouped by category. This library not only contains a rich set of functions, but also numerous types and macros.

Error Handling

Error handling in the ANSI library is done using a global variable `errno` that is set by the library routines and may be tested by a user program. There also are a few functions for error handling:

```
void assert (int expr);
void perror (const char *msg);
char * strerror (int errno);
```

String Handling Functions

Strings in ANSI-C always are null-terminated character sequences. The ANSI library provides the following functions to manipulate such strings:

```
size_t strlen (const char *s);  
char * strcpy (char *to, const char *from);  
char * strncpy (char *to, const char *from, size_t size);  
char * strcat (char *to, const char *from);  
char * strncat (char *to, const char *from, size_t size);  
int strcmp (const char *p, const char *q);  
int strncmp (const char *p, const char *q, size_t size);  
char * strchr (const char *s, int ch);  
char * strrchr (const char *s, int ch);  
char * strstr (const char *p, const char *q);  
size_t strspn (const char *s, const char *set);  
size_t strcspn (const char *s, const char *set);  
char * strupr (const char *s, const char *set);  
char * strtok (char *s, const char *delim);
```

Memory Block Functions

Closely related to the string handling functions are those operating on memory blocks. The main difference to the string functions is that they operate on any block of memory, whether it is null-terminated or not. The length of the block must be given as an additional parameter. Also, the functions work with `void` pointers instead of `char` pointers.

```
void * memcpy (void *to, const void *from, size_t size);  
void * memmove (void *to, const void *from, size_t size);  
int memcmp (const void *p, const void *q, size_t size);  
void * memchr (const void *adr, int byte, size_t size);  
void * memset (void *adr, int byte, size_t size);
```

Mathematical Functions

The ANSI library contains a variety of floating point functions. The standard interface, which is defined for type `double`, has been augmented by an alternate interface (and implementation) using type `float`.

```
double acos    (double x);
double asin    (double x);
double atan    (double x);
double atan2   (double x, double y);
double ceil    (double x);
double cos     (double x);
double cosh   (double x);
double exp     (double x);
double fabs   (double x);
double floor   (double x);
double fmod   (double x, double y);
double frexp  (double x, int *exp);
double ldexp  (double x, int exp);
double log    (double x);
double log10  (double x);
double modf   (double x, double *ip);
double pow    (double x, double y);
double sin    (double x);
double sinh   (double x);
double sqrt   (double x);
double tan    (double x);
double tanh   (double x);
```

The functions using type `float` have the same names with an “f” appended.

```
float acosf    (float x);
float asinf    (float x);
float atanf    (float x);
float atan2f   (float x, float y);
float ceilf    (float x);
float cosf     (float x);
float coshf   (float x);
float expf     (float x);
float fabsf   (float x);
float floorf   (float x);
float fmodf   (float x, float y);
float frexpf  (float x, int *exp);
float ldexpf   (float x, int exp);
float logf     (float x);
float log10f   (float x);
float modff   (float x, float *ip);
float powf     (float x, float y);
float sinf    (float x);
float sinhf   (float x);
float sqrtf   (float x);
float tanf    (float x);
float tanhf   (float x);
```

Additionally, the ANSI library also defines a couple of functions operating on integral values:

```
int abs    (int i);
div_t div   (int a, int b);
long labs  (long l);
ldiv_t ldiv (long a, long b);
```

Furthermore, it contains a simple pseudo random number generator:

```
int    rand (void);  
void   srand (unsigned int seed);
```

Memory Management

To allocate and deallocate memory blocks, the ANSI library provides the following functions:

```
void * malloc (size_t size);  
void * calloc (size_t n, size_t size);  
void * realloc (void *ptr, size_t size);  
void   free     (void *ptr);
```

Since it is not possible to implement these functions in a way that suits all possible target processors and memory configurations, these functions all are based on system module `heap.c`, which can be modified by the user to fit a particular memory layout.

Searching and Sorting

The ANSI library contains both a generalized searching and a generalized sorting procedure:

```
void * bsearch (const void *key, const void *array,  
                  size_t n, size_t size, cmp_func f);  
  
void   qsort(void *array, size_t n, size_t size, cmp_func f);
```

Character Functions

These functions test or convert characters. All these functions are implemented both as macros and as functions, by default, the macros are active. To use the corresponding function, you have to #undefine the macro.

```
int isalnum    (int ch);
int isalpha    (int ch);
int iscntrl   (int ch);
int isdigit   (int ch);
int isgraph   (int ch);
int islower   (int ch);
int isprint   (int ch);
int ispunct   (int ch);
int isspace   (int ch);
int isupper   (int ch);
int isxdigit  (int ch);
int tolower   (int ch);
int toupper   (int ch);
```

The ANSI library also defines an interface for multibyte and wide characters. The Implementation only offers minimum support for this feature: the maximum length of a multibyte character is one byte.

```
int      mblen     (char *mbs, size_t n);
size_t   mbstowcs (wchar_t *wcs, const char *mbs, size_t n);
int      mbtowc    (wchar_t *wc, const char *mbc, size_t n);
size_t   wcstombs  (char *mbs, const wchar_t *wcs size_t n);
int      wctomb    (char *mbc, wchar_t wc);
```

System Functions

The ANSI standard includes some system functions for raising and responding to signals, non-local jumping and so on.

```
void      abort    (void);
int       atexit   (void(*func) (void));
void      exit     (int status);
char     * getenv   (const char *name);
int       system   (const char *cmd);
int       setjmp   (jmp_buf env);
void      longjmp  (jmp_buf env, int val);
_sig_func signal   (int sig, _sig_func handler);
int       raise    (int sig);
```

To process variable length argument lists, the ANSI library provides the following “functions” (they’re implemented as macros):

```
void  va_start (va_list args, param);
type  va_arg   (va_list args, type);
void  va_end   (va_list args);
```

Time Functions

In the ANSI library, there also are several function to get the current time. In an embedded systems environment, we cannot provide implementations for these functions since different targets may use different ways to count the time.

```

clock_t      clock      (void);
time_t       time       (time_t *time_val);
struct tm * localtime (const time_t *time_val);
time_t       mktime     (struct tm *time_rec);
char        * asctime    (const struct tm *time_rec);
char        * ctime      (const time *time_val);
size_t       strftime   (char *s, size_t n,
                           const char *format,
                           const struct tm *time_rec);
double      difftime  (time_t t1, time_t t2);
struct tm * gmtime    (const time_t *time_val);

```

Locale Functions

These functions are for handling locales. The ANSI-C library only supports the minimal “C” environment.

```

struct lconv *localeconv (void);
char        *setlocale   (int cat, const char *locale);
int         strcoll    (const char *p, const char *q);
size_t      strxfrm(const char *p, const char *q, size_t n);

```

Conversion Functions

Functions for converting strings to numbers are

```
int      atoi      (const char *s);
long     atol      (const char *s);
double   atof      (const char *s);
long     strtol    (const char *s, char **end, int base);
unsigned long strtoul (const char *s, char **end, int base);
double   strtod    (const char *s, char **end);
```

printf and scanf

More conversions are possible for the C functions for reading and writing formatted data. These functions are:

```
int  sprintf  (char *s, const char *format, ...);
int  vsprintf (char *s, const char *format, va_list args);
int  sscanf   (const char *s, const char *format, ...);
```

File I/O

The ANSI-C library contains a fairly large interface for file I/O. In micro-controller applications however, one usually doesn't need file I/O. In the few cases where one would need it, the implementation depends on the actual setup of the target system. It

is therefore impossible for us to provide an implementation for these features, the user has to implement them her/himself.

```
FILE* fopen    (const char *name, const char *mode);
FILE* freopen  (const char *name, const char *mode, FILE *f);
int   fflush   (FILE *f);
int   fclose   (FILE *f);
int   feof     (FILE *f);
int   ferror  (FILE *f);
void  clearerr(FILE *f);
int   remove   (const char *name);
int   rename   (const char *old, const char *new);
FILE* tmpfile  (void);
char* tmpnam  (char *name);
void  setbuf   (FILE *f, char *buf);
int   setvbuf  (FILE *f, char *buf, int mode, size_t size);
```

Functions for writing and reading characters:

```
int   fgetc   (FILE *f);
char* fgets   (char *s, int n, FILE *f);
int   fputc   (int c, FILE *f);
int   fputs   (const char *s, FILE *f);
int   getc     (FILE *f);
int   getchar  (void);
char* gets    (char *s);
int   putc    (int c, FILE *f);
int   putchar  (int c);
int   puts    (const char *s);
int   ungetc  (int c, FILE *f);
```

Functions for reading and writing blocks of data:

```
size_t fread   (void *buf, size_t size, size_t n, FILE *f);
size_t fwrite  (void *buf, size_t size, size_t n, FILE *f);
```

Formatted I/O functions on files:

```
int fprintf    (FILE *f, const char *format, ...);  
int vfprintf   (FILE *f, const char *format, va_list args);  
int fscanf    (FILE *f, const char *format, ...);  
int printf     (const char *format, ...);  
int vprintf   (const char *format, va_list args);  
int scanf     (const char *format, ...);
```

Positioning functions:

```
int fgetpos   (FILE *f, fpos_t *pos);  
int fsetpos   (FILE *f, const fpos_t *pos);  
int fseek     (FILE *f, long offset, int mode);  
long ftell    (FILE *f);  
void rewind   (FILE *f);
```

Types and Macros in the Standard Library

This section discusses all types and macros defined in the ANSI standard library. We cover each of the header files, in alphabetical order.

errno.h

This header file just declared two constants, that are used as error indicators in global variable errno.

```
extern int errno;  
  
#define EDOM      -1  
#define ERANGE    -2
```

float.h

Defines constants describing the properties of floating point arithmetic. See [Table 3.2](#) and [Table 3.3](#).

Table 3.2 Rounding and Radix Constants

Constant	Description
FLT_ROUNDS	Gives the rounding mode implemented
FLT_RADIX	The base of the exponent

All other constants are prefixed by either FLT_, DBL_ or LDBL_. FLT_ is a constant for type float, DBL_ one for double and LDBL_ one for long double.

Table 3.3 Other Constants Defined in float.h

Constant	Description
DIG	Number of significant digits.
EPSILON	Smallest positive x for which $1.0 + x \neq x$.
MANT_DIG	Number of binary mantissa digits.
MAX	Largest normalized finite value.
MAX_EXP	Maximum exponent such that $\text{FLT_RADIX}^{\text{MAX_EXP}}$ is a finite normalized value.
MAX_10_EXP	Maximum exponent such that $10^{\text{MAX_10_EXP}}$ is a finite normalized value.
MIN	Smallest positive normalized value.
MIN_EXP	Smallest negative exponent such that $\text{FLT_RADIX}^{\text{MIN_EXP}}$ is a normalized value.
MIN_10_EXP	Smallest negative exponent such that $10^{\text{MIN_10_EXP}}$ is a normalized value.

limits.h

Defines a couple of constants for the maximum and minimum values that are allowed for certain types. See [Table 3.4](#).

Table 3.4 Constants Defined in limits.h

Constant	Description
CHAR_BIT	Number of bits in a character
SCHAR_MIN	Minimum value for signed char
SCHAR_MAX	Maximum value for signed char
UCHAR_MAX	Maximum value for unsigned char
CHAR_MIN	Minimum value for char
CHAR_MAX	Maximum value for char
MB_LEN_MAX	Maximum number of bytes for a multi-byte character.
SHRT_MIN	Minimum value for short int
SHRT_MAX	Maximum value for short int
USHRT_MAX	Maximum value for unsigned short int
INT_MIN	Minimum value for int
INT_MAX	Maximum value for int
UINT_MAX	Maximum value for unsigned int
LONG_MIN	Minimum value for long int
LONG_MAX	Maximum value for long int
ULONG_MAX	Maximum value for unsigned long int

locale.h

The header file in [Listing 3.1](#) defines a struct containing all the locale specific values.

Listing 3.1 Local-specific Values

```
struct lconv {           /* "C" locale (default) */
    char *decimal_point; /* "." */
    /* Decimal point character to use for non-monetary numbers */
    char *thousands_sep; /* "" */
}
```

```

/* Character to use to separate digit groups in
   the integral part of a non-monetary number. */
char *grouping;           /* "\CHAR_MAX" */

/* Number of digits that form a group. CHAR_MAX
   means "no grouping", '\0' means take previous
   value.
   E.g. the string "\3\0" specifies the repeated
   use of groups of three digits. */
char *int_curr_symbol;    /* "" */

/* 4-character string for the international
   currency symbol according to ISO 4217. The
   last character is the separator
   between currency symbol and amount. */
char *currency_symbol;    /* "" */

/* National currency symbol. */
char *mon_decimal_point;  /* "." */

char *mon_thousands_sep;  /* "" */

char *mon_grouping;       /* "\CHAR_MAX" */

/* Same as decimal_point etc., but for monetary numbers. */
char *positive_sign;     /* "" */

/* String to use for positive monetary numbers.*/
char *negative_sign;     /* "" */

/* String to use for negative monetary numbers. */
char int_frac_digits;    /* CHAR_MAX */

/* Number of fractional digits to print in a
   monetary number according to international format. */
char frac_digits;         /* CHAR_MAX */

/* The same for national format. */
char p_cs_precedes;      /* 1 */

/* 1 indicates that the currency symbol is left
   of a positive monetary amount; 0 indicates it's on the right. */
char p_sep_by_space;      /* 1 */

/* 1 indicates that the currency symbol is
   separated from the number by a space for
   positive monetary amounts. */
char n_cs_precedes;       /* 1 */

char n_sep_by_space;       /* 1 */

/* The same for negative monetary amounts. */
char p_sign_posn;         /* 4 */

char n_sign_posn;          /* 4 */

/* Defines the position of the sign for positive
   and negative monetary numbers:
   0 amount and currency are in parentheses
   1 sign comes before amount and currency
   2 sign comes after the amount

```

```

3 sign comes immediately before the currency
4 sign comes immediately after the currency */
} ;

```

There also are several constants that can be used in [setlocale](#) to define which part of the locale should be set. See [Table 3.5](#).

Table 3.5 Constants Used with setlocal()

Constant	Description
LC_ALL	Changes the complete locale.
LC_COLLATE	Only changes the locale for functions strcoll and strxfrm .
LC_MONETARY	Changes the locale for formatting monetary numbers.
LC_NUMERIC	Changes the locale for numeric, i.e. non-monetary formatting.
LC_TIME	Changes the locale for function strftime .
LC_TYPE	Changes the locale for character handling and multi-byte character functions.

This implementation only supports the minimum “C” locale.

math.h

Defines just this constant:

HUGE_VAL

Large value that is returned if overflow occurs.

setjmp.h

Contains just this type definition:

`typedef jmp_buf;`

A buffer for [setjmp](#) to store the current program state.

signal.h

Defines signal handling constants and types. See [Table 3.6](#) and [Table 3.7](#).

```
typedef sig_atomic_t;
```

Table 3.6 Constants Defined in signal.h

Constant	Definition
SIG_DFL	If passed as 2 nd argument to signal, the default response is installed.
SIG_ERR	Return value of signal , if the handler could not be installed.
SIG_IGN	If passed as 2nd argument to signal , the signal is ignored.

Table 3.7 Signal Type Constants

Constant	Definition
SIGABRT	Abort program abnormally
SIGFPE	Floating point error
SIGILL	Illegal instruction
SIGINT	Interrupt
SIGSEGV	Segmentation violation
SIGTERM	Terminate program normally

stddef.h

Defines a few generally useful types and constants. See [Table 3.8](#).

Table 3.8 Constants Defined in stddef.h

Constant	Description
ptrdiff_t	The result type of the subtraction of two pointers.
size_t	Unsigned type for the result of sizeof.
wchar_t	Integral type for wide characters.

Table 3.8 Constants Defined in stddef.h

Constant	Description
#define NULL ((void *) 0)	
size_t offsetof (type, struct_member)	Returns the offset of field struct_member in struct type.

stdio.h

There are two type declarations in this header file. See [Table 3.9](#).

Table 3.9 Type Definitions in stdio.h

Type Definition	Description
FILE	Defines a type for a file descriptor.
fpos_t	A type to hold the position in the file as needed by fgetpos and fsetpos .

[Table 3.10](#) lists the constants defined in stdio.h.

Table 3.10 Constants Defined in stdio.h

Constant	Description
BUFSIZ	Buffer size for setbuf .
EOF	Negative constant to indicate end-of-file.
FILENAME_MAX	Maximum length of a file name.
FOPEN_MAX	Maximum number of open files.
_IOFBF	To set full buffering in setvbuf .
_IOLBF	To set line buffering in setvbuf .
_IONBF	To switch off buffering in setvbuf .
SEEK_CUR	fseek positions relative from current position.
SEEK_END	fseek positions from the end of the file.
SEEK_SET	fseek positions from the start of the file.
TMP_MAX	Maximum number of unique filenames tmpnam can generate.

Additionally, there are three variables for the standard I/O streams:

```
extern FILE *stderr, *stdin, *stdout;
```

stdlib.h

Besides a redefinition of `NULL`, `size_t` and `wchar_t`, this header file contains the type definitions listed in [Table 3.11](#).

Table 3.11 Type Definitions in stdlib.h

Type Definition	Description
<code>typedef div_t;</code>	A struct for the return value of div .
<code>typedef ldiv_t;</code>	A struct for the return value of ldiv .

[Table 3.12](#) lists the constants defined in stdlib.h

Table 3.12 Constants Defined in stdlib.h

Constant	Definition
<code>EXIT_FAILURE</code>	Exit code for unsuccessful termination.
<code>EXIT_SUCCESS</code>	Exit code for successful termination.
<code>RAND_MAX</code>	Maximum return value of rand .
<code>MB_LEN_MAX</code>	Maximum number of bytes in a multi byte character.

time.h

This header files defines types and constants for time management. See [Listing 3.2](#).

Listing 3.2 time.h—Type Definitions and Constants

```
typedef clock_t;
typedef time_t;

struct tm {
    int tm_sec;           /* Seconds */
    int tm_min;           /* Minutes */
    int tm_hour;          /* Hours */
    int tm_mday;          /* Day of month: 0 .. 31 */
    int tm_mon;           /* Month: 0 .. 11 */
```

```
int tm_year;           /* Year since 1900 */
int tm_wday;          /* Day of week: 0 .. 6 (Sunday == 0) */
int tm_yday;          /* day of year: 0 .. 365 */
int tm_isdst;         /* Daylight saving time flag:
                         > 0      It is DST
                         0       It isn't DST
                         < 0      unknown */

};
```

The constant `CLOCKS_PER_SEC` gives the number of clock ticks per second.

string.h

The file `string.h` defines only functions and not types or special defines.

The functions are explained below together with all other ANSI functions.

assert.h

The file `assert.h` defines the macro [assert](#). If the macro `NDEBUG` is defined, then `assert` does nothing. Otherwise `assert` calls the auxiliary function `_assert` if the one macro parameter of `assert` evaluates to 0 (FALSE).

```
#ifdef NDEBUG
    #define assert(EX)
#else
    #define assert(EX) ((EX) ? 0 : _assert(__LINE__, __FILE__))
#endif
```

stdarg.h

The file `stdarg.h` defines the type `va_list` and the macros [va_arg](#), [va_end](#) and [va_start](#). The type `va_list` implements a pointer to one argument of a open parameter list. The macro [va_start](#) initializes a variable of type `va_list` to point to the first open parameter, given the last explicit parameter and its type as arguments. The macro [va_arg](#) returns one open parameter, given its type and also makes the `va_list` argument pointing to the next parameter. The [va_end](#) macro finally releases the actual pointer. For all implementations, the [va_end](#) macro does nothing because `va_list` is implemented as elementary data type and it must therefore not be released. The [va_start](#) and the [va_arg](#) macro have a type parameter, which is accessed only with `sizeof`. So type, but also variables can be used.

Look at the following example:

```
char sum(long p, ...) {
    char res=0;
    va_list list= va\_start(p, long);
    res= va\_arg(list, int);      // (*)
    va\_end(list);
    return res;
}
void main(void) {
    char c=2;
    if (f(10L, c) != 2) Error();
}
```

In the line (*) [va_arg](#) must be called with int, not with char. Because of the default argument promotion rules of C, for integral types at least an int is passed and for floating types at least a double is passed. In other words, the result of using [va_arg](#)(..., char) or [va_arg](#)(..., short) is undefined in C. Especially be careful when using variables instead of types for [va_arg](#). In the example above “res= [va_arg](#)(list, res)” would not be correct unless res would have the type int and not char.

ctype.h

The ctype.h file defines functions to check properties of characters, as if a character is a digit ([isdigit](#)), a space ([isspace](#)) and many others. These functions are either implemented as macros, or as real functions. The macro version is used when the compiler option [-Ot](#) is used or the macro [__OPTIMIZE_FOR_TIME__](#) is defined. The macros use a table of 257 bytes length called [_ctype](#). In this array, all properties tested by the various functions are encoded by single bits, taking the character as indices into the array. The function implementations otherwise do not use this table. They save memory by using the shorter call to the function (compared with the expanded macro).

The functions in [Listing 3.3](#) are explained below together with all other ANSI functions.

Listing 3.3 Macros Defined in ctype.h

```
extern unsigned char _ctype[];
#define _U (1<<0)      /* Upper case      */
#define _L (1<<1)      /* Lower case      */
#define _N (1<<2)      /* Numeral (digit) */
#define _S (1<<3)      /* Spacing character */
#define _P (1<<4)      /* Punctuation     */
#define _C (1<<5)      /* Control character */
```

```
#define _B  (1<<6)      /* Blank           */
#define _X  (1<<7)      /* heXadecimal digit */

#ifndef __OPTIMIZE_FOR_TIME__ /* -Ot defines this macro */
#define isalnum(c)  (_ctype[(unsigned char)(c+1)] & (_U|_L|_N))
#define isalpha(c)  (_ctype[(unsigned char)(c+1)] & (_U|_L))
#define iscntrl(c)  (_ctype[(unsigned char)(c+1)] & _C)
#define isdigit(c)  (_ctype[(unsigned char)(c+1)] & _N)
#define isgraph(c)  (_ctype[(unsigned char)(c+1)] & (_P|_U|_L|_N))
#define islower(c)  (_ctype[(unsigned char)(c+1)] & _L)
#define isprint(c)  (_ctype[(unsigned char)(c+1)] &(_P|_U|_L|_N|_B))
#define ispunct(c)  (_ctype[(unsigned char)(c+1)] & _P)
#define isspace(c)  (_ctype[(unsigned char)(c+1)] & _S)
#define isupper(c)  (_ctype[(unsigned char)(c+1)] & _U)
#define isxdigit(c) (_ctype[(unsigned char)(c+1)] & _X)
#define tolower(c)  (isupper(c) ? ((c) - 'A' + 'a') : (c))
#define toupper(c)  (islower(c) ? ((c) - 'a' + 'A') : (c))
#define isascii(c)  (!((c) & ~127))
#define toascii(c)  (c & 127)
#endif /* __OPTIMIZE_FOR_TIME__ */
```

The Standard Functions

This section describes all the standard functions in the ANSI-C library. Each function description contains the subsections listed in [Table 3.13](#).

Table 3.13 Function Description Subsections

Subsection	Description
Syntax	Shows the function's prototype and also which header file to include.
Description	A description of how to use the function.
Return	Describes what the function returns in which case. If global variable <code>errno</code> is modified by the function, possible values are described, too.
See also	Contains cross-references to related functions.

Functions not implemented because the implementation would be hardware specific anyway (e.g. [clock](#)) are marked by:

*Hardware
specific*



appearing in the right margin next to the function's name. Functions for file I/O, which also depend on your particular hardware's setup and therefore also are not implemented, are marked by:

File I/O



in the right margin.

abort

Syntax

```
#include <stdlib.h>
void abort (void);
```

Description

`abort()` terminates your program. It does the following (in this order):

- raises signal `SIGABRT`
- flushes all open output streams
- closes all open files
- removes all temporary files
- calls `HALT`

If your application handles `SIGABRT` and the signal handler doesn't return (e.g. because it does a [longjmp](#)), the application is not halted.

See also

[atexit](#), [exit](#), [raise](#), [signal](#)

abs

Syntax

```
#include <stdlib.h>
int abs (int i);
```

Description

`abs()` computes the absolute value of `i`.

Return

The absolute value of `i`, i.e. `i` if `i` is positive and `-i` if `i` is negative. If `i` is `-32768`, this value is returned and `errno` is set to `ERANGE`.

See also

[fabs](#), [labs](#)

acos, acosf

Syntax

```
#include <math.h>
double acos (double x);
float acosf (float x);
```

Description

`acos()` computes the principal value of the arc cosine of `x`.

Return

The arc cosine $\cos^{-1}(x)$ of `x` in the range between 0 and Pi if `x` is in the range $-1 \leq x \leq 1$. If `x` is not in this range, NAN is returned and `errno` is set to `EDOM`.

See also

[asin](#), [asinf](#), [atan](#), [atanf](#), [atan2](#), [atan2f](#), [cos](#), [cosf](#), [sin](#), [sinf](#),
[tan](#), [tanf](#)

asctime

*Hardware
specific*



Syntax

```
#include <time.h>
char * asctime (const struct tm* timeptr);
```

Description

`asctime()` converts the broken down time in `timeptr` into a string.

Return

A pointer to a string containing the time string.

See also

[localtime](#), [mktme](#), [time](#)

asin, asinf

Syntax

```
#include <math.h>
double asin (double x);
float asinf (float x);
```

Description

`asin()` computes the principal value of the arc sine of `x`.

Return

The arc sine $\sin^{-1}(x)$ of x in the range between $-\pi/2$ and $\pi/2$ if x is in the range $-1 \leq x \leq 1$. If x is not in this range, `NAN` is returned and `errno` is set to `EDOM`.

See also

[acos](#), [acosf](#), [atan](#), [atanf](#), [atan2](#), [atan2f](#), [cos](#), [cosf](#), [sin](#), [sinf](#), [tan](#), [tanf](#)

assert

Syntax

```
#include <assert.h>
void assert (int expr);
```

Description

`assert()` is a macro that indicates expression `expr` is expected to be true at this point in the program. If `expr` is false (0), `assert` halts the program.

Compiling with option `-DNDEBUG` or placing the preprocessor control statement

```
#define NDEBUG
```

before the `#include <assert.h>` statement effectively deletes all assertions from the program.

See also

[abort](#), [exit](#)

atan, atanf

Syntax

```
#include <math.h>
double atan (double x);
float atanf (float x);
```

Description

`atan()` computes the principal value of the arc tangent of x .

Return

The arc tangent $\tan^{-1}(x)$, in the range from $-\pi/2$ to $\pi/2$ rad.

See also

[acos](#), [acosf](#), [asin](#), [asinf](#), [atan2](#), [atan2f](#), [cos](#), [cosf](#), [sin](#), [sinf](#), [tan](#), [tanf](#)

atan2, atan2f

Syntax

```
#include <math.h>
double atan2 (double y, double x);
float atan2f (float y, float x);
```

Description

`atan2()` computes the principal value of the arc tangent of y/x . It uses the sign of both operands to determine the quadrant of the result.

Return

The arc tangent $\tan^{-1}(y/x)$, in the range from $-\pi$ to π rad, if not both x and y are 0. If both x and y are 0, it returns 0.

See also

[acos](#), [acosf](#), [asin](#), [asinf](#), [atan](#), [atanf](#), [cos](#), [cosf](#), [sin](#), [sinf](#), [tan](#), [tanf](#)

atexit

Syntax

```
#include <stdlib.h>
int atexit (void (*func) (void));
```

Description

`atexit()` lets you install a function that is to be executed just before the normal termination of the program. You can register at most 32 functions with `atexit`. These functions are called in the reverse order they were registered.

Return

`atexit` returns 0 if it could register the function, otherwise it returns a non-zero value.

See also

[abort](#), [exit](#)

atof

Syntax

```
#include <stdlib.h>
double atof (const char *s);
```

Description

`atof()` converts the string `s` to a double floating point value, skipping over white space at the beginning of `s`. It stops converting when it reaches either the end of the string or a character that cannot be part of the number. The number format accepted by `atof` is the following:

FloatNum	= Sign {Digit} ["."] {Digit} [Exp].
Sign	= ["+" "-"].
Digit	= <any decimal digit from 0 to 9>.
Exp	= ("e" "E") Sign Digit {Digit}.

Return

atof returns the converted double floating point value.

See also

[atoi](#), [atol](#), [strtod](#), [strtol](#), [strtoul](#)

atoi

Syntax

```
#include <stdlib.h>
int atoi (const char *s);
```

Description

atoi() converts the string s to an integer value, skipping over white space at the beginning of s. It stops converting when it reaches either the end of the string or a character that cannot be part of the number. The number format accepted by atoi is the following:

Number = ["+"|"-"] Digit {Digit}.

Return

atoi returns the converted integer value.

See also

[atof](#), [atol](#), [strtod](#), [strtol](#), [strtoul](#)

atol

Syntax

```
#include <stdlib.h>
long atol (const char *s);
```

Description

`atol()` converts the string `s` to an `long` value, skipping over white space at the beginning of `s`. It stops converting when it reaches either the end of the string or a character that cannot be part of the number. The number format accepted by [atoi](#) is the following:

Number = ["+"|"-"] Digit {Digit}.

Return

`atol` returns the converted `long` value.

See also

[atoi](#), [atof](#), [strtod](#), [strtol](#), [strtoul](#)

bsearch

Syntax

```
#include <stdlib.h>

void *bsearch (const void *key,
               const void *array,
               size_t n,
               size_t size,
               cmp_func cmp);
```

Description

`bsearch()` performs a binary search in a sorted array. It calls the comparison function `cmp` with two arguments: a pointer to the key element that is to be found and a pointer to an array element. Thus, the type `cmp_func` can be declared as

```
typedef int (*cmp_func)(const void *key, const void *data);
```

The comparison function should return an integer as follows:

If the key element is...	the return value should be...
less than the array element	less than zero (negative)
equal to the array element	zero
greater than the array element	greater than zero (positive)

The arguments to bsearch() are:

Parameter Name	Meaning
key	A pointer to the key data you are looking for
array	A pointer to the beginning (i.e. the first element) of the array you want to search
n	The number of elements in the array
size	The size (in bytes) of one element in the table
cmp	The comparison function

NOTE Make sure the array contains only elements of the same size.
bsearch also assumes that the array is sorted in ascending order with respect to the comparison function cmp.

Return

bsearch() returns a pointer to an element of the array that matches the key, if there is one. If the comparison function never returns zero, i.e. there is no matching array element, bsearch() returns NULL.

calloc

Hardware specific



Syntax

```
#include <stdlib.h>
void *calloc (size_t n, size_t size);
```

Description

calloc() allocates a block of memory for an array containing n elements of size size. All bytes in the memory block are initialized to zero. To deallocate the block, use [free\(\)](#). The default implementation is not reentrant and should therefore not be used in interrupt routines.

Return

`calloc` returns a pointer to the allocated memory block. If the block couldn't be allocated, the return value is `NULL`.

See also

[free](#), [malloc](#), [realloc](#)

ceil, ceilf

Syntax

```
#include <math.h>  
  
double ceil (double x);  
float ceilf (float x);
```

Description

`ceil()` returns the smallest integral number larger than `x`.

See also

[floor](#), [floorf](#), [modf](#), [modff](#)

clearerr

File I/O



Syntax

```
#include <stdio.h>  
  
void clearerr (FILE *f);
```

Description

`clearerr()` resets the error flag and the EOF marker of file `f`.

clock

Hardware
specific



Syntax

```
#include <time.h>
clock_t clock (void);
```

Description

`clock()` determines the amount of time since your system started, in clock ticks. To convert to seconds, divide by `CLOCKS_PER_SEC`.

Return

`clock` returns the amount of time since system startup.

See also

[time](#)

cos, cosf

Syntax

```
#include <math.h>
double cos (double x);
float cosf (float x);
```

Description

`cos()` computes the principal value of the cosine of `x`. `x` should be expressed in radians.

Return

The cosine `cos(x)`.

See also

[acos](#), [acosf](#), [asin](#), [asinf](#), [atan](#), [atanf](#), [atan2](#), [atan2f](#), [sin](#), [sinf](#), [tan](#), [tanf](#)

cosh, coshf

Syntax

```
#include <math.h>  
  
double cosh (double x);  
float coshf (float x);
```

Description

`cosh()` computes the hyperbolic cosine of `x`.

Return

The hyperbolic cosine `cosh(x)`. If the computation fails because the value is too large, `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

See also

[cos](#), [cosf](#), [sinh](#), [sinhf](#), [tanh](#), [tanhf](#)

ctime

Hardware specific



Syntax

```
#include <time.h>  
  
char *ctime (const time_t *timer);
```

Description

`ctime()` converts the calendar time timer to a character string.

Return

The string containing the ASCII representation of the date.

See also

[asctime](#), [mktime](#), [time](#)

difftime

*Hardware
specific*



Syntax

```
#include <time.h>
double difftime (time_t *t1, time_t t0);
```

Description

`difftime()` calculates the number of seconds between any two calendar times.

Return

The number of seconds between the two times, as a double.

See also

[mktme](#), [time](#)

div

Syntax

```
#include <stdlib.h>
div_t div (int x, int y);
```

Description

`div()` computes both the quotient and the modulus of the division x/y .

Return

A structure with the results of the division.

See also

[ldiv](#)

exit

Syntax

```
#include <stdlib.h>
void exit (int status);
```

Description

`exit()` terminates your program normally. It does the following, in this order:

- executes all functions registered with [atexit](#)
- flushes all open output streams
- closes all open files
- removes all temporary files
- calls HALT

The `status` argument is ignored.

See also

[abort](#), [atexit](#)

exp, expf

Syntax

```
#include <math.h>
double exp (double x);
float expf (float x);
```

Description

`exp()` computes e^x , where e is the base of natural logarithms.

Return

e^x . If the computation fails because the value is too large, `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

See also

[log](#), [logf](#), [log10](#), [log10f](#), [pow](#), [powf](#)

fabs, fabsf

Syntax

```
#include <math.h>
double fabs (double x);
float fabsf (float x);
```

Description

`fabs()` computes the absolute value of `x`.

Return

The absolute value of `x` for any value of `x`.

See also

[abs](#), [labs](#)

fclose

File I/O



Syntax

```
#include <stdio.h>
int fclose (FILE *f);
```

Description

`fclose()` closes file `f`. Before doing so, it does the following:

- flushes the stream, if the file wasn't opened in read-only mode
- discards and deallocates any buffers that were allocated automatically, i.e. not using [setbuf](#).

Return

Zero, if the function succeeds, EOF otherwise.

See also

[fopen](#), [setbuf](#)

feof

File I/O



Syntax

```
#include <stdio.h>
int feof (FILE *f);
```

Description

`feof()` tests whether previous I/O calls on file `f` tried to do anything beyond the end of the file.

NOTE

Calling [clearerr](#) or [fseek](#) clears the file's end-of-file flag, therefore `feof` returns 0.

Return

Zero, if we're not at the end of the file, EOF otherwise.

See also

[clearerr](#), [fseek](#)

ferror

File I/O



Syntax

```
#include <stdio.h>
int ferror (FILE *f);
```

Description

`ferror()` tests whether an error had occurred on file `f`. To clear the error indicator of a file, use `clearerr`. `rewind` automatically resets the file's error flag.

NOTE Do not use `ferror` to test for end-of-file. Use `feof` instead.

Return

Zero, if there was no error, non-zero otherwise.

See also

[clearerr](#), [feof](#), [rewind](#)

fflush

File I/O



Syntax

```
#include <stdio.h>
int fflush (FILE *f);
```

Description

`fflush()` flushes the I/O buffer of file `f`, letting you switch cleanly between reading and writing the same file. If your program was writing to file `f`, `fflush` writes all buffered data to the file, if it was reading, `fflush` discards any buffered data. If `f` is `NULL`, *all* files open for writing are flushed.

Return

Zero, if there was no error, EOF otherwise.

See also

[setbuf](#), [setvbuf](#)

fgetc

File I/O



Syntax

```
#include <stdio.h>
int fgetc (FILE *f);
```

Description

`fgetc()` reads the next character from file `f`.

NOTE If file `f` had been opened as a text file, the end-of-line character combination is read as one '`\n`' character.

Return

The character read as an integer in the range from 0 to 255. If there was a read error, `fgetc` returns `EOF` and sets the file's error flag, so that a subsequent call to `ferror` will return a non-zero value. If you try to read beyond the end of the file, `fgetc` also returns `EOF` but sets the end-of-file flag instead of the error flag, so that `feof` will return `EOF`, but `ferror` will return 0.

See also

[feof](#), [ferror](#), [fgets](#), [fopen](#), [fread](#), [fscanf](#), [getc](#)

fgetpos

File I/O



Syntax

```
#include <stdio.h>
int fgetpos (FILE *f, fpos_t *pos);
```

Description

`fgetpos()` returns in `*pos` the current file position. You can use this value to later set the position to this one using [fsetpos](#).

NOTE	Do <i>not</i> assume the value in <code>*pos</code> to have any particular meaning such as a byte offset from the beginning of the file. The ANSI standard does not require this, and in fact any value may be put into <code>*pos</code> as long as a fsetpos with that value resets the position in the file correctly.
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Return

Non-zero, if there was an error; zero otherwise.

See also

[fseek](#), [fsetpos](#), [ftell](#)

fgets

File I/O



Syntax

```
#include <stdio.h>
char *fgets (char *s, int n, FILE *f);
```

Description

`fgets()` reads a string of at most $n-1$ characters from file `f` into `s`. Immediately after the last character read, a '`\0`' is appended. If `fgets` reads a line break ('`\n`') or reaches the end of the file before having read $n-1$ characters, the following happens:

- If `fgets` reads a line break, it adds the '`\n`' plus a '`\0`' to `s` and returns successfully.
- If it reaches the end of the file after having read at least 1 character, it adds a '`\0`' to `s` and returns successfully.
- If it reaches EOF without having read any character, it sets the file's end-of-file flag and returns unsuccessfully. (`s` is left unchanged).

Return

`NULL`, if there was an error; `s` otherwise.

See also

[fgetc](#), [fputs](#)

floor, floorf

Syntax

```
#include <math.h>
double floor (double x);
float floorf (float x);
```

Description

`floor()` calculates the largest integral number not larger than `x`.

Return

The largest integral number not larger than `x`.

See also

[ceil](#), [ceilf](#), [modf](#), [modff](#)

fmod, fmodf

Syntax

```
#include <math.h>
double fmod (double x, double y);
float fmodf (float x, float y);
```

Description

`fmod()` calculates the floating point remainder of `x/y`.

Return

The floating point remainder of `x/y`, with the same sign as `x`. If `y` is 0, it returns 0 and sets `errno` to `EDOM`.

See also

[div](#), [ldiv](#), [ldexp](#), [ldexpf](#), [modf](#), [modff](#)

fopen

File I/O



Syntax

```
#include <stdio.h>
FILE *fopen (const char *name, const char *mode);
```

Description

`fopen()` opens a file with the given name and mode. It automatically allocates an I/O buffer for the file.

There are three main modes: read, write, and update (i.e. both read and write) accesses. Each can be combined with either text or binary mode to read a text file or update a binary file. Opening a file for text accesses translates the end-of-line character (combination) into '`\n`' when reading and vice versa when writing. The table below lists all possible modes.

Mode	Effect
"r"	Open the file as text file for reading.
"w"	Create a text file and open it for writing.
"a"	Open the file as text file for appending
"rb"	Open the file as binary file for reading.
"wb"	Create a file and open as binary file for writing.
"ab"	Open the file as binary file for appending.
"r+"	Open a text file for updating.
"w+"	Create a text file and open for updating.
"a+"	Open a text file for updating. Append all writes to the end.
"r+b", "rb+"	Open a binary file for updating.
"w+b", "wb+"	Create a binary file and open for updating.
"a+b", "ab+"	Open a binary file for updating, appending all writes to the end.

If the mode contains an “r”, but the file doesn’t exist, fopen returns unsuccessfully. Opening a file for appending (mode contains “a”) always appends writing to the end, even if you call [fseek](#), [fsetpos](#) or [rewind](#). Opening a file for updating allows both read and write accesses on the file. However, you must call [fseek](#), [fsetpos](#) or [rewind](#) to write after a read or to read after a write.

Return

A pointer to the file descriptor of the file. If the file could not be created, the function returns NULL.

See also

[fclose](#), [freopen](#), [fseek](#), [fsetpos](#), [rewind](#), [setbuf](#), [setvbuf](#)

fprintf

File I/O



Syntax

```
#include <stdio.h>
int fprintf (FILE *f, const char *format, ...);
```

Description

`fprintf()` is the same as [sprintf](#), but the output goes to file `f` instead of a string.

For a detailed format description see [sprintf](#).

Return

The number of characters written. If some error occurred, EOF is returned.

See also

[printf](#), [sprintf](#), [vsprintf](#)

fputc

File I/O



Syntax

```
#include <stdio.h>
int fputc (int ch, FILE *f);
```

Description

`fputc()` writes a character to file `f`.

Return

The integer value of `ch`. If an error occurred, `fputc` returns EOF.

See also

[fputs](#)

fputs

File I/O



Syntax

```
#include <stdio.h>
int fputs (const char *s, FILE *f);
```

Description

`fputs()` writes the zero-terminated string `s` to file `f` (without the terminating '`\0`'.

Return

EOF, if there was an error, zero otherwise.

See also

[fputc](#)

fread

File I/O



Syntax

```
#include <stdio.h>
size_t fread (void *ptr, size_t size, size_t n, FILE *f);
```

Description

`fread()` reads a contiguous block of data. It attempts to read `n` items of size `size` from file `f` and stores them in the array `ptr` points to. If either `n` or `size` is 0, nothing is read from the file and the array is left unchanged.

Return

The number of items successfully read.

See also

[fgetc](#), [fgets](#), [fwrite](#)

free

Hardware specific



Syntax

```
#include <stdlib.h>
void free (void *ptr);
```

Description

`free()` deallocates a memory block that had previously been allocated by [calloc](#), [malloc](#) or [realloc](#). If `ptr` is NULL, nothing happens. The default implementation is not reentrant and should therefore not be used in interrupt routines.

See also

[calloc](#), [malloc](#), [realloc](#)

freopen

File I/O



Syntax

```
#include <stdio.h>
void freopen (const char *name,
              const char *mode,
              FILE *f);
```

Description

`freopen()` opens a file using a specific file descriptor. This can be useful for redirecting `stdin`, `stdout` or `stderr`. About possible modes, see [fopen](#).

See also

[fopen](#), [fclose](#)

frexp, frexpf

Syntax

```
#include <math.h>
double frexp (double x, int *exp);
float frexpf (float x, int *exp);
```

Description

`frexp()` splits a floating point number into mantissa and exponent. The relation is $x = m \cdot 2^{\text{exp}}$. m always is normalized to the range $0.5 < m \leq 1.0$. The mantissa has the same sign as x .

Return

The mantissa of x (the exponent is written to `*exp`). If x is 0.0 , both the mantissa (the return value) and the exponent are 0 .

See also

[exp](#), [expf](#), [ldexp](#), [ldexpf](#), [modf](#), [modff](#)

fscanf

File I/O



Syntax

```
#include <stdio.h>
int fscanf (FILE *f, const char *format, ...);
```

Description

`fscanf()` is the same as [sscanf](#), but the input comes from file f instead of a string.

Return

The number of data arguments read, if any input was converted. If not, it returns EOF.

See also

[fgetc](#), [fgets](#), [scanf](#), [sscanf](#)

fseek

File I/O



Syntax

```
#include <stdio.h>
int fseek (FILE *f, long offset, int mode);
```

Description

`fseek()` sets the current position in file f.

For binary files, the position can be set in three ways, as shown in this table.

mode	Position is set to...
SEEK_SET	offset bytes from the beginning of the file.
SEEK_CUR	offset bytes from the current position.
SEEK_END	offset bytes from the end of the file.

For text files, either `offset` must be zero or `mode` is `SEEK_SET` and `offset` a value returned by a previous call to [f.tell](#).

If `fseek` is successful, it clears the file's end-of-file flag. You cannot set the position beyond the end of the file.

Return

Zero, if successful; non-zero otherwise.

See also

[f.getpos](#), [f.setpos](#), [f.tell](#)

f.setpos

File I/O



Syntax

```
#include <stdio.h>
int f.setpos (FILE *f, const fpos_t *pos);
```

Description

`f.setpos()` set the file position to `pos`, which must be a value returned by a previous call to [f.getpos](#) on the same file. If the function is successful, it clears the file's end-of-file flag.

You cannot set the position beyond the end of the file.

Return

Zero, if it was successful, non-zero otherwise.

See also

[f.getpos](#), [f.seek](#), [f.tell](#)

f~~tell~~

File I/O



Syntax

```
#include <stdio.h>
long ftell (FILE *f);
```

Description

`ftell()` returns the current file position. For binary files, this is the byte offset from the beginning of the file; for text files, this value should not be used except as argument to [fseek](#).

Return

–1, if an error occurred; otherwise the current file position.

See also

[fgetpos](#), [fseek](#), [fsetpos](#)

fwrite

File I/O



Syntax

```
#include <stdio.h>
size_t fwrite (const void *p,
              size_t size,
              size_t n,
              FILE *f);
```

Description

`fwrite()` writes a block of data to file `f`. It writes `n` items of size `size`, starting at address `ptr`.

Return

The number of items successfully written.

See also

[fputc](#), [fputs](#), [fread](#)

getc

File I/O



Syntax

```
#include <stdio.h>
int getc (FILE *f);
```

Description

getc() is the same as [fgetc](#), but may be implemented as a macro. Therefore, you should make sure that f is not an expression having side effects! See [fgetc](#) for more information.

getchar

File I/O



Syntax

```
#include <stdio.h>
int getchar (void);
```

Description

getchar() is the same as [getc](#)(stdin). See [fgetc](#) for more information.

getenv

File I/O



Syntax

```
#include <stdio.h>
char *getenv (const char *name);
```

Description

`getenv()` returns the value of environment variable name.

Return

NULL

gets

File I/O



Syntax

```
#include <stdio.h>
char *gets (char *s);
```

Description

`gets()` reads a string from `stdin` and stores it in `s`. It stops reading when it reaches a line break or `EOF` character. This character is not appended to the string. The string is zero-terminated.

If the function reads `EOF` before any other character, it sets `stdin`'s end-of-file flag and returns unsuccessfully without changing string `s`.

Return

NULL, if there was an error; `s` otherwise.

See also

[fgetc](#), [puts](#)

gmtime

Hardware specific



Syntax

```
#include <time.h>
struct tm *gmtime (const time_t *time);
```

Description

`gmtime()` converts `*time` to UTC (Universal Coordinated Time), which is equivalent to GMT (Greenwich Mean Time).

Return

`NULL`, if UTC is not available; a pointer to a struct containing UTC otherwise.

See also

[ctime](#), [time](#)

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

Syntax

```
#include <ctype.h>
int isalnum  (int ch);
int isalpha  (int ch);
...
int isxdigit (int ch);
```

Description

These functions determine whether character `ch` belongs to a certain set of characters. The next table describes the character ranges tested by the functions.

Function	Tests whether <code>ch</code> is in the range...
<code>isalnum</code>	alphanumeric character, i.e. ' <code>'A'-'Z'</code> ', ' <code>'a'-'z'</code> ' or ' <code>'0'-'9'</code> '.
<code>isalpha</code>	an alphabetic character, i.e. ' <code>'A'-'Z'</code> ' or ' <code>'a'-'z'</code> '.
<code>iscntrl</code>	a control character, i.e. ' <code>\000'-'037'</code> or ' <code>\177</code> ' (DEL).
<code>isdigit</code>	a decimal digit, i.e. ' <code>'0'-'9'</code> '.
<code>isgraph</code>	a printable character except space (' <code>!'-'~'</code>).
<code>islower</code>	a lower case letter, i.e. ' <code>'a'-'z'</code> '.
<code>isprint</code>	a printable character (' <code>-'-'~'</code>).

Function	Tests whether ch is in the range...
ispunct	a punctuation character, i.e. '!'-'/', ':'-'@', '['-'{' and '{'-'~'.
isspace	a white space character, i.e. ' ', '\f', '\n', '\r', '\t' and '\v'.
isupper	an upper case letter, i.e. 'A'-'Z'.
isxdigit	a hexadecimal digit, i.e. '0'-'9', 'A'-'F' or 'a'-'f'.

Return

TRUE (i.e. 1), if ch is in the character class; zero otherwise.

See also

[tolower](#), [toupper](#)

labs

Syntax

```
#include <stdlib.h>
long labs (long i);
```

Description

`labs()` computes the absolute value of `i`.

Return

The absolute value of `i`, i.e. `i` if `i` is positive and `-i` if `i` is negative. If `i` is `-2147483648`, this value is returned and `errno` is set to `ERANGE`.

See also

[abs](#), [fabs](#)

ldexp, ldexpf

Syntax

```
#include <math.h>
double ldexp (double x, int exp);
float ldexpf (float x, int exp);
```

Description

`ldexp()` multiplies x by 2^{exp} .

Return

$x * 2^{\text{exp}}$. If it fails because the result would be too large, `HUGE_VAL` is returned and `errno` is set to `ERANGE`.

See also

[exp](#), [expf](#), [frexp](#), [frexpf](#), [log](#), [logf](#), [log10](#), [log10f](#), [mod](#), [modf](#),
[fabs](#)

ldiv

Syntax

```
#include <stdlib.h>
ldiv_t ldiv (long x, long y);
```

Description

`ldiv()` computes both the quotient and the modulus of the division x/y .

Return

A structure with the results of the division.

See also

[div](#)

localeconv

*Hardware
specific*



Syntax

```
#include <locale.h>
struct lconv *localeconv (void);
```

Description

`localeconv()` returns a pointer to a `struct` containing information about the current locale, e.g. how to format monetary quantities.

Return

A pointer to a `struct` containing the desired information.

See also

[setlocale](#)

localtime

*Hardware
specific*



Syntax

```
#include <time.h>
struct tm *localtime (const time_t *time);
```

Description

`localtime()` converts `*time` into broken-down time.

Return

A pointer to a `struct` containing the broken-down time.

See also

[asctime](#), [mktime](#), [time](#)

log, logf

Syntax

```
#include <math.h>
double log (double x);
float logf (float x);
```

Description

`log()` computes the natural logarithm of `x`.

Return

`ln (x)`, if `x` is greater than zero. If `x` is smaller then zero, `NAN` is returned, if it's equal zero, `log` returns negative infinity. In both cases, `errno` is set to `EDOM`.

See also

[exp](#), [expf](#), [log10](#), [log10f](#)

log10, log10f

Syntax

```
#include <math.h>
double log10 (double x);
float log10f (float x);
```

Description

`log10()` computes the decadic logarithm (the logarithm to base 10) of `x`.

Return

`log10(x)`, if `x` is greater than zero. If `x` is smaller then zero, `NAN` is returned, if it's equal zero, `log10` returns negative infinity. In both cases, `errno` is set to `EDOM`.

See also

[exp](#), [expf](#), [log](#), [logf](#)

longjmp

Syntax

```
#include <setjmp.h>
void longjmp ( jmp_buf env, int val);
```

Description

`longjmp()` performs a non-local jump to some location earlier in the call chain. That location must have been marked by a call to [setjmp](#). The environment at the time of that call to [setjmp](#) (`env`, which also must have been the parameter to [setjmp](#)) is restored and your application continues as if the call to [setjmp](#) just had returned value `val`.

See also

[setjmp](#)

malloc

Hardware specific



Syntax

```
#include <stdlib.h>
void *malloc (size_t size);
```

Description

`malloc()` allocates a block of memory for an object of size `size` bytes. The content of this memory block is undefined. To deallocate the block, use [free\(\)](#). The default implementation is not reentrant and should therefore not be used in interrupt routines.

Return

`malloc` returns a pointer to the allocated memory block. If the block couldn't be allocated, the return value is `NULL`.

See also

[calloc](#), [free](#), [realloc](#)

mblen

Hardware specific



Syntax

```
#include <stdlib.h>
int mbstrlen (const char *s, size_t n);
```

Description

`mblen()` determines the number of bytes the multi-byte character pointed to by `s` occupies.

Return

0, if `s` is `NULL`.
-1, if the first `n` bytes of `*s` do not form a valid multi-byte character.
`n`, the number of bytes of the multi-byte character otherwise.

See also

[mbtowc](#), [mbstowcs](#)

mbstowcs

Hardware specific



Syntax

```
#include <stdlib.h>
size_t mbstowcs (wchar_t *wcs,
                  const char *mbs,
                  size_t n);
```

Description

`mbstowcs()` converts a multi-byte character string `mbs` to a wide character string `wcs`. Only the first `n` elements are converted.

Return

The number of elements converted, or `(size_t) -1` if there was an error.

See also

[mblen](#), [mbtowc](#)

mbtowc

Hardware specific



Syntax

```
#include <stdlib.h>
int mbtowc (wchar_t *wc, const char *s, size_t n);
```

Description

`mbtowc()` converts a multi-byte character `s` to a wide character code `wcs`. Only the first `n` bytes of `*s` are taken into consideration.

Return

The number of bytes of the multi-byte character converted, or `(size_t) -1` if there was an error.

See also

[mblen](#), [mbstowcs](#)

memchr

Syntax

```
#include <string.h>
void *memchr (const void *p, int ch, size_t n);
```

Description

`memchr()` looks for the first occurrence of a byte containing (`ch & 0xFF`) in the first `n` bytes of the memory are pointed to by `p`.

Return

A pointer to the byte found, or `NULL` if no such byte was found.

See also

[memcmp](#), [strchr](#), [strrchr](#)

memcmp

Syntax

```
#include <string.h>
void *memcmp (const void *p,
              const void *q,
              size_t n);
```

Description

`memcmp()` compares the first `n` bytes of the two memory areas pointed to by `p` and `q`.

Return

A positive integer, if `p` is considered greater than `q`; a negative integer if `p` is considered smaller than `q` and zero if the two memory area are equal.

See also

[memchr](#), [strcmp](#), [strncmp](#)

memcpy, memmove

Syntax

```
#include <string.h>

void *memcpy  (const void *p,
               const void *q,
               size_t n);
void *memmove (const void *p,
               const void *q,
               size_t n);
```

Description

Both functions copy *n* bytes from *q* to *p*. *memmove* also works if the two memory areas overlap.

Return

p.

See also

[strcpy](#), [strncpy](#)

memset

Syntax

```
#include <string.h>

void *memset (void *p, int val, size_t n);
```

Description

memset() sets the first *n* bytes of the memory area pointed to by *p* to the value (*val* & 0xFF).

Return

p.

See also

[calloc](#), [memmove](#)

mktime

Hardware specific



Syntax

```
#include <time.h>
time_t mktime (struct tm *time);
```

Description

`mktime()` converts `*time` to a `time_t`. The fields of `*time` may have any value, they're not restricted to the ranges given [time.h](#). If the conversion was successful, `mktime` restricts the fields of `*time` to these ranges and also sets the fields `tm_wday` and `tm_yday` correctly.

Return

`*time` as a `time_t`.

See also

[ctime](#), [gmtime](#), [time](#)

modf, modff

Syntax

```
#include <math.h>
double modf  (double x, double *i);
float  modff (float x, float *i);
```

Description

`modf()` splits the floating point number `x` into integral part (returned in `*i`) and fractional part. Both parts have the same sign as `x`.

Return

The fractional part of x .

See also

[floor](#), [floorf](#), [fmod](#), [fmodf](#), [frexp](#), [frexpf](#), [ldexp](#), [ldexpf](#)

perror

Syntax

```
#include <stdio.h>
void perror (const char *msg);
```

Description

`perror()` writes an error message appropriate for the current value of `errno` to `stderr`. The character string `msg` is part of `perror`'s output.

See also

[assert](#), [strerror](#)

pow, powf

Syntax

```
#include <math.h>
double pow (double x, double y);
float powf (float x, float y);
```

Description

`pow()` computes x to the power of y , i.e. x^y .

Return

x^y , if $x > 0$
1, if $y == 0$

+x, if ($x == 0 \&& y < 0$)
NAN, if ($x < 0 \&& y$ is not integral). Also, `errno` is set to `EDOM`.
±x, with the same sign as x , if the result is too large.

See also

[exp](#), [expf](#), [ldexp](#), [ldexf](#), [log](#), [logf](#), [modf](#), [modff](#)

printf

File I/O



Syntax

```
#include <stdio.h>
int printf (const char *format, ...);
```

Description

`printf()` is the same as [sprintf](#), but the output goes to `stdout` instead of a string.

For a detailed format description see [sprintf](#).

Return

The number of characters written. If some error occurred, `EOF` is returned.

See also

[fprintf](#), [sprintf](#), [vsprintf](#)

putc

File I/O



Syntax

```
#include <stdio.h>
int putc (char ch, FILE *f);
```

Description

`putc()` is the same as [fputc](#), but may be implemented as a macro. Therefore, you should make sure that `f` is not an expression having side effects! See [fputc](#) for more information.

See also

[fputc](#)

putchar

File I/O



Syntax

```
#include <stdio.h>
int putchar (char ch);
```

Description

`putchar(ch)` is the same as [putc\(ch, stdin\)](#). See “[fputc](#)” for more information.

See also

[fputc](#)

puts

File I/O



Syntax

```
#include <stdio.h>
int puts (const char *s);
```

Description

`puts()` writes string `s` followed by a newline '\n' to `stdout`.

Return

EOF, if there was an error; zero otherwise.

See also

[fputc](#), [putc](#)

qsort

Syntax

```
#include <stdlib.h>
void *qsort (const void *array,
             size_t n,
             size_t size,
             cmp_func cmp);
```

Description

`qsort()` sorts the array according to the ordering implemented by the comparison function. It calls the comparison function `cmp` with two pointers to array elements. Thus, the type `cmp_func` can be declared as

```
typedef int (*cmp_func)(const void *key,
                        const void *other);
```

The comparison function should return an integer as follows:

If the key element is...	The return value should be...
less than the other one	less than zero (negative)
equal to the other one	zero
greater than the other one	greater than zero (positive)

The arguments to `qsort` are:

Argument Name	Meaning
<code>array</code>	A pointer to the beginning (i.e. the first element) of the array you want to sort
<code>n</code>	The number of elements in the array
<code>size</code>	The size (in bytes) of one element in the table
<code>cmp</code>	The comparison function

NOTE Make sure the array contains elements of the same size.

raise

Syntax

```
#include <signal.h>
int raise (int sig);
```

Description

`raise()` raises the given signal, invoking your signal handler or performing the defined response to the signal. If you didn't define a response or install a signal handler, your application is aborted.

Return

Non-zero, if there was an error, zero otherwise.

See also

[signal](#)

rand

Syntax

```
#include <stdlib.h>
int rand (void);
```

Description

`rand()` generates a pseudo random number in the range from 0 to `RAND_MAX`. The numbers are generated are based on a seed, which initially is 1. To change the seed, use [`srand`](#).

Equal seeds always lead to the same sequence of pseudo random numbers.

Return

A pseudo random integer in the range from 0 to RAND_MAX.

See also

[srand](#)

realloc

Hardware
specific



Syntax

```
#include <stdlib.h>
void *realloc (void *ptr, size_t size);
```

Description

`realloc()` changes the size of a block of memory, preserving its contents. `ptr` must be a pointer returned by [calloc](#), [malloc](#) or `realloc`, or `NULL`. In the latter case, `realloc` is equivalent to [malloc](#).

If the new size of the memory block is smaller than the old size, `realloc` discards that memory at the end of the block. If `size` is zero (and `ptr` not `NULL`), `realloc` frees the whole memory block.

If there's not enough memory to perform the `realloc`, the old memory block is left unchanged and `realloc` returns `NULL`. The default implementation is not reentrant and should therefore not be used in interrupt routines.

Return

`realloc` returns a pointer to the new memory block. If the operation couldn't be performed, the return value is `NULL`.

See also

[calloc](#), [free](#), [malloc](#)

remove

File I/O



Syntax

```
#include <stdio.h>
int remove (const char *filename);
```

Description

`remove()` deletes the file `filename`. If the file is open, `remove` doesn't delete it and returns unsuccessfully.

Return

Non-zero, if there was an error; zero otherwise.

See also

[tmpfile](#), [tmpnam](#)

rename

File I/O



Syntax

```
#include <stdio.h>
int rename (const char *from, const char *to);
```

Description

`rename()` renames file `from` to `to`. If there already is a file `to`, `rename` doesn't change anything and returns with an error code.

Return

Non-zero, if there was an error; zero otherwise.

See also

[tmpfile](#), [tmpnam](#)

rewind

File I/O



Syntax

```
#include <stdio.h>
void rewind (FILE *f);
```

Description

`rewind()` resets the current position in file f to the beginning of the file. It also clears the file's error indicator.

See also

[fopen](#), [fseek](#), [fsetpos](#)

scanf

File I/O



Syntax

```
#include <stdio.h>
int scanf (const char *format, ...);
```

Description

`scanf()` is the same as [sscanf](#), but the input comes from `stdin` instead of a string.

Return

The number of data arguments read, if any input was converted. If not, it returns EOF.

See also

[fgetc](#), [fgets](#), [fscanf](#), [sscanf](#)

setbuf

File I/O



Syntax

```
#include <stdio.h>
void setbuf (FILE *f, char *buf);
```

Description

`setbuf()` lets you specify how a file is buffered. If `buf` is `NULL`, the file is unbuffered, i.e. all input or output goes directly to and comes directly from the file. If `buf` is not `NULL`, it is used as a buffer (`buf` should point to an array of `BUFSIZ` bytes).

See also

[fflush](#), [setvbuf](#)

setjmp

Syntax

```
#include <setjmp.h>
int setjmp (jmp_buf env);
```

Description

`setjmp()` saves the current program state in environment buffer `env` and returns zero. This buffer can be used as parameter to a later call to [longjmp](#), which then restores the program state and jumps back to the location of the `setjmp`. This time, `setjmp` returns a non-zero value, which is equal to the second parameter to [longjmp](#).

Return

Zero if called directly, non-zero if called by a [longjmp](#).

See also

[longjmp](#)

setlocale

*Hardware
specific*



Syntax

```
#include <locale.h>
char *setlocale (int class, const char *loc);
```

Description

`setlocale()` changes the program's locale – either all or just part of it, depending on `class`. The new locale is given by the character string `loc`. The classes allowed are given by the table below.

Class	Changes the locale...
LC_ALL	for all classes.
LC_COLLATE	for the functions <code>strcoll</code> and <code>strxfrm</code> .
LC_MONETARY	for monetary formatting.
LC_NUMERIC	for numeric formatting.
LC_TIME	for function <code>strftime</code> .
LC_TYPE	for character handling and multi-byte character functions.

HI-CROSS supports only the minimum locale “C” (see [locale.h](#)), so this function has no effect.

Return

”C”, if `loc` is “C” or NULL; NULL otherwise.

See also

[localeconv](#), [strcoll](#), [strftime](#), [strxfrm](#)

setvbuf

File I/O



Syntax

```
#include <stdio.h>
void setvbuf (FILE *f,
              char *buf,
              int mode,
              size_t size);
```

Description

`setvbuf()` lets you specify how a file is buffered. `mode` determines how the file is buffered.

Mode	Buffering
<code>_IOFBF</code>	Fully buffered
<code>_IOLBF</code>	Line buffered
<code>_IONBF</code>	Unbuffered

To make a file unbuffered, call `setvbuf` with mode `_IONBF`; the other arguments (`buf` and `size`) are ignored.

In all other modes, the file uses buffer `buf` of size `size`. If `buf` is `NULL`, the function allocates a buffer of size `size` itself.

See also

[fflush](#), [setbuf](#)

signal

Syntax

```
#include <signal.h>
_sig_func signal (int sig, _sig_func handler);
```

Description

`signal()` defines how your application shall respond to signal `sig`. The various responses are given by the table below.

Handler	Response to the signal
<code>SIG_IGN</code>	The signal is ignored.
<code>SIG_DFL</code>	The default response (HALT).
a function	The function is called with <code>sig</code> as parameter.

The signal handling function is defined as

```
typedef void (*_sig_func) (int sig);
```

The signal can be raised using function [raise](#). Before the handler is called, the response is reset to `SIG_DFL`.

In HI-CROSS, there are only two signals: `SIGABRT` indicates an abnormal program termination, and `SIGTERM` a normal program termination.

Return

If signal succeeds, it returns the previous response for the signal, otherwise it returns `SIG_ERR` and sets `errno` to a positive non-zero value.

See also

[raise](#)

sin, sinf

Syntax

```
#include <math.h>
double sin (double x);
float sinf (float x);
```

Description

`sin()` computes the sine of `x`.

Return

The sine $\sin(x)$ of x in radians.

See also

[asin](#), [asinf](#), [acos](#), [acosf](#), [atan](#), [atanf](#), [atan2](#), [atan2f](#), [cos](#), [cosf](#), [tan](#), [tanf](#)

sinh, sinhf

Syntax

```
#include <math.h>  
  
double sinh (double x);  
float  sinhf (float x);
```

Description

$\sinh()$ computes the hyperbolic sine of x .

Return

The hyperbolic sine $\sinh(x)$ of x . If it fails because the value is too large, it returns infinity with the same sign as x and set `errno` to `ERANGE`.

See also

[asin](#), [asinf](#), [cosh](#), [coshf](#), [sin](#), [sinf](#), [tan](#), [tanf](#)

sprintf

Syntax

```
#include <stdio.h>  
  
int sprintf (char *s, const char *format, ...);
```

Description

`sprintf()` writes formatted output to string `s`. It evaluates the arguments, converts them according to format and writes the result to `s`, terminated with a zero character.

The format string contains the text to be printed. Any character sequence in format starting with '%' is a format specifier that are replaced by the corresponding argument. The first format specifier is replace with the first argument after format, the second format specifier by the second argument, and so on.

A format specifier has the form

`FormatSpec` = "%" {Format} [Width] [Precision]
[Length] Conversion.

`Format` = "-" | "+" | " " | "#" | "0".

The format defines justification and sign information (the latter only for numerical arguments). A "-" left justifies the output, a "+" forces output of the sign, and a blank output a blank if the number is positive and a "-" if it's negative. The effect of "#" depends on the conversion character following.

Conversion	Effect of "#"
e, E, f	The value of the argument always is printed with decimal point, even if there are no fractional digits.
g, G	As above, but additionally zeroes are appended to the fraction until the specified width is reached.
o	A zero is printed before the number to indicate an octal value.
x, X	"0x" (if the conversion is "x") or "0X" (if it is "X") is printed before the number to indicate a hexadecimal value.
others	undefined.

A "0" as format specifier adds leading zeroes to the number until the desired width is reached, if the conversion character specifies a numerical argument.

If both " " and "+" are given, only "+" is active; if both "0" and "-" are specified, only "-" is active. If there's a precision specification for integral conversions, "0" is ignored.

`Width` = "*" | Number.

The width defines the minimum field width into which the output is to be put. If the argument is smaller, space is filled as defined by the format characters.

If a "*" is given, the field width is taken from the next argument, which of course must be a number. If that number is negative, output is left justified.

Precision = ". " [Number | "*"] .

The effect of the precision specification depends on the conversion character.

Conversion	Precision
d, i, o, u, x, X	The minimum number of digits to print.
e, E, f	The number of fractional digits to print.
g, G	The maximum number of significant digits to print.
s	The maximum number of characters to print.
others	undefined.

If the precision specifier is "*", the precision is taken from the next argument, which must be an int. If that value is negative, the precision is ignored.

Length = "h" | "l" | "L" .

A length specifier tells `sprintf`, what type the argument has. The first two length specifiers can be used in connection with all conversion characters for integral numbers. "h" defines short, "l" defines long. Specifier "L" is used in conjunction with the conversion characters for floating point numbers and specifies long double.

Conversion = "c" | "d" | "e" | "E" | "f" | "g" | "G" | "i" | "n" | "o" | "p" | "s" | "u" | "x" | "X" | "%" .

These conversion characters have the following meanings:

Conversion	Description
c	The int argument is converted to unsigned char, the resulting character is printed.
d, i	An int argument is printed.
e, E	The argument must be a double. It is printed in the form [-]d.ddde±dd (scientific notation). The precision determines the number of fractional digits, the digit to the left of the decimal is 0 unless the argument is 0.0. Default precision is 6 digits. If precision is zero and the format specifier "#" is not given, no decimal point is printed. The exponent always has at least 2 digits; the conversion character is printed just before the exponent.

Conversion	Description
f	The argument must be a double. It's printed in the form [-]ddd.ddd. See above. If the decimal point is printed, there's at least one digit left of it.
g, G	The argument must be a double. sprintf chooses either format "f" or "e" (or "E" if "G" is given), depending on the magnitude of the value: scientific notation is used only, if the exponent is <-4 or greater or equal to the precision.
n	The argument must be a pointer to an int. sprintf writes the number of characters written so far to that address. If "n" is used together with length specifier "h" or "l", the argument must be a pointer to a short int or a long int.
o	The argument, which must be an unsigned int, is printed in octal notation.
p	The argument must be a pointer; its value is printed in hexadecimal notation.
s	The argument must be a char *; sprintf writes the string.
u	The argument, which must be an unsigned int, is written in decimal notation.
x, X	The argument, which must be an unsigned int, is written in hexadecimal notation. "x" uses lower case letters "a" to "f", while "X" uses upper case letters.
%	Prints a "%" sign. Should only be given as "%%".

Conversion characters for integral types are "d", "i", "o", "u", "x", "X"; for floating point types "e", "E", "f", "g", "G".

If sprintf finds an incorrect format specification, it stops processing, terminates the string with a zero character and returns successfully.

NOTE	Floating point support increases the sprintf size considerably and therefore the define "LIBDEF_PRINTF_FLOATING" exists which should be set if no floating point support is used in. Some targets do contain special libraries without floating point support. The IEEE64 floating point implementation only supports printing numbers with up to 9 decimal digits. This limitation occurs because the implementation is using unsigned long internally which can not
-------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

hold more digits. Supporting more digits would increase the printf size once more and would also work considerably slower.

Return

The number of characters written to `s`.

See also

[sscanf](#)

sqrt, sqrtf

Syntax

```
#include <math.h>
double sqrt (double x);
float sqrtf (float x);
```

Description

`sqrt()` computes the square root of `x`.

Return

The square root of `x`. If `x` is negative, it returns 0 and set `errno` to `EDOM`.

See also

[pow](#)

strnd

Syntax

```
#include <stdlib.h>
void strnd (unsigned int seed);
```

Description

`srand()` initializes the seed of the random number generator. The default seed is 1.

See also

[rand](#)

sscanf

Syntax

```
#include <stdio.h>
int sscanf (const char *s, const char *format, ...);
```

Description

`sscanf()` scans string `s` according to the given format, storing the values in the parameters given. Format specifiers in `format` tell `sscanf`, what to expect next. A format specifier has the format

FormatSpec = "%" [Flag] [Width] [Size] Conversion.

Flag = "*"

If the "%" sign which starts a format specification is followed by a "*", the scanned value is not assigned to the corresponding parameter.

Width = Number.

Specifies the maximum number of characters to read when scanning the value. Scanning also stops if a white space or a character not matching the expected syntax is reached.

Size = "h" | "l" | "L"

Specifies the size of the argument to read. The meaning is given in the table below:

Size	Legal Conversions	Parameter Type
h	d, i, n	short int * (instead of int *)
h	o, u, x, X	unsigned short int * (instead of unsigned int *)
l	d, i, n	long int * (instead of int *)
l	o, u, x, X	unsigned long int * (instead of unsigned int *)
l	e, E, f, g, G	double * (instead of float *)
L	e, E, f, g, G	long double * (instead of float *)

Conversion = "c" | "d" | "e" | "E" | "f" | "g" |
 | "G" | "i" | "n" | "o" | "p" | "s" |
 | "u" | "x" | "X" | "%" | Range.

These conversion characters tell `sscanf` what to read and how to store it in a parameter. Their meaning is shown in this table:

Conversion	Description
c	Read a string of exactly width characters and stores it in the parameter. If no width is given, one character is read. The argument must be a <code>char *</code> . The string read is <i>not</i> zero-terminated.
d	A decimal number (syntax below) is read and stored in the parameter. The parameter must be a pointer to an integral type.
i	As "d", but also reads octal and hexadecimal numbers (syntax below).
e, E, f, g, G	Reads a floating point number (syntax below). The parameter must be a pointer to a floating point type.
n	The argument must be a pointer to an <code>int</code> . <code>sscanf</code> writes the number of characters read so far to that address. If "n" is used together with length specifier "h" or "l", the argument must be a pointer to a <code>short int</code> or a <code>long int</code> .
o	Reads an octal number (syntax below). The parameter must be a pointer to an integral type.
p	Reads a pointer in the same format as printf prints it. The parameter must be a <code>void **</code> .

Conversion	Description
s	Reads a character string up to the next white space character or at most width characters. The string is zero-terminated. The argument must be of type <code>char *</code> .
u	As "d", but the parameter must be a pointer to an unsigned integral type.
x, X	As "u", but reads a hexadecimal number.
%	Skips a "%" sign in the input. Should only be given as "%%".

```

Range          = "[" [ "^" ] List "]"
List           = Element {Element}
Element        = <any char> [ "-" <any char>]

```

You can also use a scan set to read a character string that either contains only the given characters, or that contains only characters not in the set. A scan set always is bracketed by left and right brackets, if the first character in the set is "^", the set is inverted (i.e. only characters *not* in the set are allowed). You can specify whole character ranges, e.g. "A-Z" specifies all upper case letters. If you want to include a right bracket in the scan set, it must be the first element in the list, a dash ("−") must be either the first or the last element. A "^" that shall be included in the list instead of indicating an inverted list must not be the first character after the left bracket. Some examples are:

```

[A-Za-z] Allows all upper and lower case characters.
[^A-Z] Allows any character that's not an
         upper case character.
[]abc] Allows "]", "a", "b" and "c".
[^]abc] Allows any char except "]", "a", "b" and "c".
[-abc] Allows "-", "a", "b" and "c".

```

A white space in the format string skips all white space characters up to the next non-white-space character. Any other character in the format must be exactly matched by the input, otherwise `sscanf` stops scanning.

The syntax for numbers as scanned by `sscanf` is the following:

```

Number      = FloatNumber | IntNumber.
IntNumber   = DecNumber | OctNumber | HexNumber.
DecNumber   = Sign Digit {Digit}.
OctNumber   = Sign "0" {OctDigit}.
HexNumber   = "0" ("x"|"X") HexDigit {HexDigit}.
FloatNumber = Sign {Digit} ["."] {Digit} [Exponent].

```

```
Exponent      = ( "e" | "E" ) DecNumber .
OctDigit      = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" .
Digit          = OctDigit | "8" | "9" .
HexDigit       = Digit | "A" | "B" | "C" | "D" | "E" | "F" |
                  "a" | "b" | "c" | "d" | "e" | "f"
```

Return

EOF, if *s* is NULL; otherwise it returns the number of arguments filled in.

NOTE If `sscanf` finds an illegal input (i.e. not matching the required syntax), it simply stops scanning and returns successfully!

See also

[sprintf](#)

strcat

Syntax

```
#include <string.h>
char *strcat (char *p, const char *q);
```

Description

`strcat()` appends string *q* to the end of string *p*. Both strings and the resulting concatenation are zero-terminated.

Return

p

See also

[memcpy](#), [memmove](#), [strcpy](#), [strncat](#), [strncpy](#)

strchr

Syntax

```
#include <string.h>
char *strchr (const char *p, int ch);
```

Description

`strchr()` looks for character `ch` in string `p`. If `ch` is '\0', the function looks for the end of the string.

Return

A pointer to the character, if found; if there's no such character in `*p`, `NULL` is returned.

See also

[memchr](#), [strrchr](#), [strstr](#)

strcmp

Syntax

```
#include <string.h>
int strcmp (const char *p, const char *q);
```

Description

`strcmp()` compares the two strings, using the character ordering given by the ASCII character set.

Return

A negative integer, if `p` is smaller than `q`; zero, if both strings are equal, and a positive integer if `p` is greater than `q`.

NOTE

The return values of `strcmp` are such that it could be used as comparison function in [bsearch](#) and [qsort](#).

See also

[memcmp](#), [strcoll](#), [strncmp](#)

strcoll**Syntax**

```
#include <string.h>
int strcoll (const char *p, const char *q);
```

Description

`strcoll()` compares the two strings interpreting them according to the current locale, using the character ordering given by the ASCII character set.

Return

A negative integer, if `p` is smaller than `q`; zero, if both strings are equal, and a positive integer if `p` is greater than `q`.

See also

[memcmp](#), [strcmp](#), [strncmp](#)

strcpy**Syntax**

```
#include <string.h>
char *strcpy (char *p, const char *q);
```

Description

`strcpy()` copies string `q` into string `p` (including the terminating '\0').

Return

p

See also

[memcpy](#), [memmove](#), [strncpy](#)

strcspn

Syntax

```
#include <string.h>
size_t strcspn (const char *p, const char *q);
```

Description

strcspn() searches p for the first character that also appears in q.

Return

The length of the initial segment of p that contains only characters *not* in q.

See also

[strchr](#), [strpbrk](#), [strrchr](#), [strspn](#)

strerror

Syntax

```
#include <string.h>
char *strerror (int errno);
```

Description

strerror() returns an error message appropriate for error number errno.

Return

A pointer to the message string.

See also

[perror](#)

strftime

Syntax

```
#include <time.h>
size_t strftime (char *s,
                 size_t max,
                 const char *format,
                 const struct tm *time);
```

Description

`strftime()` converts `time` to a character string `s`. If the conversion results in a string longer than `max` characters (including the terminating '\0'), `s` is left unchanged and the function returns unsuccessfully. How the conversion is done is determined by the `format` string. This string contains text, which is copied one-to-one to `s`, and format specifiers. The latter always start with a '%' sign and are replaced by the following:

Format	Replaced with
%a	Abbreviated name of the weekday of the current locale, e.g. "Fri".
%A	Full name of the weekday of the current locale, e.g. "Friday".
%b	Abbreviated name of the month of the current locale, e.g. "Feb".
%B	Full name of the month of the current locale, e.g. "February".
%c	Date and time in the form given by the current locale.
%d	Day of the month in the range from 0 to 31.
%H	Hour, in 24-hour-clock format.
%I	Hour, in 12-hour-clock format.
%j	Day of the year, in the range from 0 to 366.

Format	Replaced with
%m	Month, as a decimal number from 0 to 12.
%M	Minutes
%p	AM/PM specification of a 12-hour clock or equivalent of current locale.
%S	Seconds
%U	Week number in the range from 0 to 53, with Sunday as the first day of the first week.
%w	Day of the week (Sunday = 0, Saturday = 6).
%W	Week number in the range from 0 to 53, with Monday as the first day of the first week.
%x	The date in format given by current locale.
%X	The time in format given by current locale.
%y	The year in short format, e.g. "93".
%Y	The year, including the century (e.g. "1993").
%Z	The time zone, if it can be determined.
%%	A single '%' sign.

Return

If the resulting string would have had more than `max` characters, zero is returned, otherwise the length of the created string is returned.

See also

[mkttime](#), [setlocale](#), [time](#)

strlen

Syntax

```
#include <string.h>
size_t strlen (const char *s);
```

Description

`strlen()` returns the number of characters in string `s`.

Return

The length of the string.

strncat

Syntax

```
#include <string.h>
char *strncat (char *p, const char *q, size_t n);
```

Description

`strncat()` appends string `q` to string `p`. If `q` contains more than `n` characters, only the first `n` characters of `q` are appended to `p`. The two strings and the result all are zero-terminated.

Return

`p`

See also

[strcat](#)

strcmp

Syntax

```
#include <string.h>
char *strcmp (char *p, const char *q, size_t n);
```

Description

`strcmp()` compares at most the first `n` characters of the two strings.

Return

A negative integer, if *p* is smaller than *q*; zero, if both strings are equal, and a positive integer if *p* is greater than *q*.

See also

[memcmp](#), [strcmp](#)

strncpy

Syntax

```
#include <string.h>
char *strncpy (char *p, const char *q, size_t n);
```

Description

strncpy() copies at most the first *n* characters of string *q* to string *p*, overwriting *p*'s previous contents. If *q* contains less than *n* characters, a '\0' is appended.

Return

p

See also

[memcpy](#), [strcpy](#)

strpbrk

Syntax

```
#include <string.h>
char *strpbrk (const char *p, const char *q);
```

Description

strpbrk() searches for the first character in *p* that also appears in *q*.

Return

NULL, if there's no such character in p; a pointer to the character otherwise.

See also

[strchr](#), [strcspn](#), [strrchr](#), [strspn](#)

strrchr

Syntax

```
#include <string.h>
char *strrchr (const char *s, int c);
```

Description

strpbrk() searches for the last occurrence of character ch in s.

Return

NULL, if there's no such character in p; a pointer to the character otherwise.

See also

[strchr](#), [strcspn](#), [strpbrk](#), [strspn](#)

strspn

Syntax

```
#include <string.h>
size_t strspn (const char *p, const char *q);
```

Description

strspn() returns the length of the initial part of p that contains only characters also appearing in q.

Return

The position of the first character in *p* that's not in *q*.

See also

[strchr](#), [strcspn](#), [strpbrk](#), [strrchr](#)

strstr

Syntax

```
#include <string.h>
char *strstr (const char *p, const char *q);
```

Description

strstr() looks for substring *q* appearing in string *p*.

Return

A pointer to the beginning of the first occurrence of string *q* in *p*, or NULL, if *q* doesn't appear in *p*.

See also

[strchr](#), [strcspn](#), [strpbrk](#), [strrchr](#), [strspn](#)

strtod

Syntax

```
#include <stdlib.h>
double strtod (const char *s, char **end);
```

Description

strtod() converts string *s* into a floating point number, skipping over any white space at the beginning of *s*. It stops scanning when it reaches a character

not matching the required syntax and returns a pointer to that character in *end. The number format `strtod` accepts is

```
FloatNum   = Sign {Digit} ["."] {Digit} ] [Exp].  
Sign       = [ "+" | "-" ].  
Exp        = ("e" | "E") Sign Digit {Digit}.  
Digit      = <any decimal digit from 0 to 9>.
```

Return

The floating point number read. If an underflow occurred, 0.0 is returned, if the value causes an overflow, `HUGE_VAL` is returned. In both cases, `errno` is set to `ERANGE`.

See also

[atof](#), [sscanf](#), [strtol](#), [strtoul](#)

strtok

Syntax

```
#include <string.h>  
char *strtok (char *p, const char *q);
```

Description

`strtok()` breaks string `p` into tokens which are separated by at least one character appearing in `q`. The first time, call `strtok` with the string to break up as first parameter, all following times, pass `NULL` as first parameter: `strtok` will continue at the position it stopped last time (`strtok` saves the string `p` if it isn't `NULL`).

NOTE

This function is not re-entrant, because it uses a global variable for saving string `p`. ANSI defines this function in this way.

Return

A pointer to the token found, or `NULL`, if no token was found.

See also

[strchr](#), [strcspn](#), [strpbrk](#), [strrchr](#), [strspn](#), [strstr](#)

strtol

Syntax

```
#include <stdlib.h>
long strtol (const char *s, char **end, int base);
```

Description

`strtol()` converts string `s` into a `long int` of base `base`, skipping over any white space at the beginning of `s`. It stops scanning when it reaches a character not matching the required syntax (or a character too large for a given base) and returns a pointer to that character in `*end`. The number format `strtol` accepts is

IntNumber	= DecNumber OctNumber HexNumber OtherNum.
DecNumber	= Sign Digit {Digit}.
OctNumber	= Sign "0" {OctDigit}.
HexNumber	= "0" ("x" "X") HexDigit {HexDigit}.
OtherNum	= Sign OtherDigit {OtherDigit}.
OctDigit	= "0" "1" "2" "3" "4" "5" "6" "7".
Digit	= OctDigit "8" "9".
HexDigit	= Digit "A" "B" "C" "D" "E" "F" "a" "b" "c" "d" "e" "f".
OtherDigit	= HexDigit <any char between 'G' and 'Z'> <any char between 'g' and 'z'>.

The base must be 0 or in the range from 2 to 36. If it's between 2 and 36, `strtol` converts a number in that base (digits larger than 9 are represented by upper or lower case characters from 'A' to 'Z'). If base is zero, the function uses the prefix to find the base. If the prefix is "0", base 8 (octal) is assumed, if it's "0x" or "0X", base 16 (hexadecimal) is taken; any other prefixes make `strtol` scan a decimal number.

Return

The number read. If no number is found, zero is returned; if the value is smaller than `LONG_MIN` or larger than `LONG_MAX`, `LONG_MIN` or `LONG_MAX` is returned and `errno` is set to `ERANGE`.

See also

[atoi](#), [atol](#), [sscanf](#), [strtod](#), [strtoul](#)

strtoul

Syntax

```
#include <stdlib.h>

unsigned long strtoul (const char *s,
                      char **end,
                      int base);
```

Description

`strtoul()` converts string `s` into an `unsigned long int` of base `base`, skipping over any white space at the beginning of `s`. It stops scanning when it reaches a character not matching the required syntax (or a character too large for a given base) and returns a pointer to that character in `*end`. The number format `strtoul` accepts is the same as for [strtol](#) except that the negative sign is not allowed, and so are the possible values for `base`.

Return

The number read. If no number is found, zero is returned; if the value is larger than `ULONG_MAX`, `ULONG_MAX` is returned and `errno` is set to `ERANGE`.

See also

[atoi](#), [atol](#), [sscanf](#), [strtod](#), [strtol](#)

strxfrm

Syntax

```
#include <string.h>
size_t strxfrm (char *p, const char *q, size_t n);
```

Description

`strxfrm()` transforms string `q` according to the current locale, such that the comparison of two strings converted with `strxfrm` using `strcmp` yields the same result as a comparison using `strcoll`. If the resulting string would be longer than `n` characters, `p` is left unchanged.

Return

The length of the converted string.

See also

[setlocale](#), [strcmp](#), [strcoll](#)

system

*Hardware
specific*



Syntax

```
#include <string.h>
int system (const char *cmd);
```

Description

`system()` executes the command line `cmd`.

Return

Zero.

tan, tanf

Syntax

```
#include <math.h>  
  
double tan (double x);  
float tanf (float x);
```

Description

`tan()` computes the tangent of `x`. `x` should be in radians.

Return

`tan(x)`. If `x` is an odd multiple of $\pi/2$, it returns infinity and sets `errno` to `EDOM`.

See also

[acos](#), [acosf](#), [asin](#), [asinf](#), [atan](#), [atanf](#), [atan2](#), [atan2f](#), [cos](#), [cosf](#), [sin](#), [sinf](#), [tanh](#), [tanhf](#)

tanh, tanhf

Syntax

```
#include <math.h>  
  
double tanh (double x);  
float tanhf (float x);
```

Description

`tanh()` computes the hyperbolic tangent of `x`.

Return

`tanh(x)`.

See also

[atan](#), [atanf](#), [atan2](#), [atan2f](#), [cosh](#), [coshf](#), [sinh](#), [sinhf](#), [tan](#), [tanf](#)

time

Hardware specific



Syntax

```
#include <time.h>
time_t time (time_t *timer);
```

Description

`time()` gets the current calendar time. If `timer` is not `NULL`, it is assigned to it.

Return

The current calendar time.

See also

[clock](#), [mktime](#), [strftime](#)

tmpfile

File I/O



Syntax

```
#include <stdio.h>
FILE *tmpfile (void);
```

Description

`tmpfile()` creates a new temporary file using mode "wb+". Temporary files automatically are deleted when they're closed or the application ends.

Return

A pointer to the file descriptor, if the file could be created, `NULL` otherwise.

See also

[fopen](#), [tmpnam](#)

tmpnam

File I/O



Syntax

```
#include <stdio.h>
char *tmpnam (char *s);
```

Description

`tmpnam()` creates a new unique file name. If `s` is not NULL, this name is assigned to it.

Return

A unique file name.

See also

[tmpfile](#)

tolower

Syntax

```
#include <ctype.h>
int tolower (int ch);
```

Description

`tolower()` converts any upper case character in the range from 'A' to 'Z' into a lower case character from 'a' to 'z'.

Return

If `ch` is an upper case character, the corresponding lower case letter. Otherwise, `ch` is returned (unchanged).

See also

[islower](#), [isupper](#), [toupper](#)

toupper

Syntax

```
#include <ctype.h>
int toupper (int ch);
```

Description

`tolower()` converts any lower case character in the range from 'a' to 'z' into an upper case character from 'A' to 'Z'.

Return

If `ch` is a lower case character, the corresponding upper case letter. Otherwise, `ch` is returned (unchanged).

See also

[islower](#), [isupper](#), [tolower](#)

ungetc

File I/O



Syntax

```
#include <stdio.h>
int ungetc (int ch, FILE *f);
```

Description

`ungetc()` pushes the single character `ch` back onto the input stream `f`. The next read from `f` will read that character.

Return

`ch`

See also

[fgetc](#), [fopen](#), [getc](#), [getchar](#)

va_arg, va_end, va_start

Syntax

```
#include <stdarg.h>

void va_start (va_list args, param);
type va_arg   (va_list args, type);
void va_end   (va_list args);
```

Description

These macros can be used to get the parameters in an open parameter list. Calls to va_arg get a parameter of the given type. The following example shows how to do it:

```
void my_func (char *s, ...) {
    va_list args;
    int      i;
    char    *q;

    va_start (args, s);
    /* First call to 'va_arg' gets the first arg. */
    i = va_arg (args, int);
    /* Second call gets the second argument. */
    q = va_arg (args, char *);
    ...
    va_end (args);
}
```

vfprintf, vprintf, vsprintf

File I/O



Syntax

```
#include <stdio.h>

int vfprintf (FILE *f,
              const char *format,
              va_list args);
int vprintf  (const char *format, va_list args);
int vsprintf (char *s,
```

```
const char *format,  
va_list args);
```

Description

These functions are the same as [fprintf](#), [printf](#) and [sprintf](#), except that they take a va_list instead of an open parameter list as argument.

For a detailed format description see [sprintf](#).

NOTE Only vsprintf is implemented, since the other two functions depend on the actual setup and environment of your target.

Return

The number of characters written, if successful; a negative number otherwise.

See also

[fprintf](#), [printf](#), [sprintf](#), [va_arg](#), [va_end](#), [va_start](#)

wctomb

Hardware specific



Syntax

```
#include <stdlib.h>  
int wctomb (char *s, wchar_t wchar);
```

Description

wctomb() converts wchar to a multi-byte character, stores that character in s and returns the length in bytes of s.

Return

The length of s in bytes after the conversion.

See also

[wcstombs](#)

wcstombs

*Hardware
specific*



Syntax

```
#include <stdlib.h>
int wcstombs (char *s, const wchar_t *ws, size_t n);
```

Description

`wcstombs()` converts the first `n` wide character codes in `ws` to multi-byte characters, stores them character in `s` and returns the number of wide characters converted.

Return

The number of wide characters converted.

See also

[wctomb](#)

compactC++ Library

C++

Introduction

This chapter describes how to use the cC++ library provided with the compiler.

The library is a subset of the ANSI-C++ standard C++ library draft. It is based on the public domain GNU C++ library (Copyright by GNU).

The STL library is the Hewlett Packard public-domain implementation (Copyright by HP).

So all sources of the library are provided with your compiler.

This chapter is only a short introduction into the C++ library. For further informations please refer to the book "The Draft Standard C++ library" by P.J.Plauger (Prentice Hall, 1995)

Implementation

The cC++ library consists of following source files:

- `streamb.h`, `streamb.cpp`
- `iostream.h`, `iostream.cpp`
- `cstring.h`, `cstring.cpp`
- `new.h`, `new.cpp`
- `stderr.cpp`
- `builtin.cpp`
- `regex.cpp`
- `stdstr.cpp`
- `stdstrb.cpp`
- `error.cpp`

- `complex.h`, `complext.cpp`
- `bitset.h`, `bitset.cpp`
- `bitstr.cpp`
- `strstrea.h`, `strstrea.cpp`
- `iopadn.c`
- `iofflush.c`
- `genops.c`
- `bitand.c`
- `bitany.c`
- `bitblt.c`
- `bitclear.c`
- `bitcopy.c`
- `bitcount.c`
- `bitinv.c`
- `bitlcomp.c`
- `bitset1.c`
- `bitxor.c`
- `strops.c`
- `stdiostr.h`

If you write C++ programs needing the C++ library, you have to specify the name of the library built with the desired memory model and options in the “NAMES” section of the link parameter file (*.prm) of your project.

I/O Library

Description

This library component makes input and output look simple. It mainly consists of 4 objects:

- `cin`
To control unbuffered standard input stream (as does `stdin` of the ANSI-C library)
- `cout`

To control unbuffered standard output stream (as does `stdout` of the ANSI-C library)

- `cerr`
To control unbuffered standard error stream (as does `stderr` of the ANSI-C library)
- `clog`
To control buffered standard error stream (as does `std::log` of the ANSI-C library)

These objects offer an easy way to perform input and output in a C++ program.

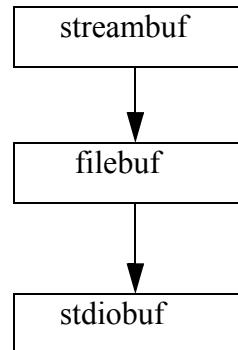
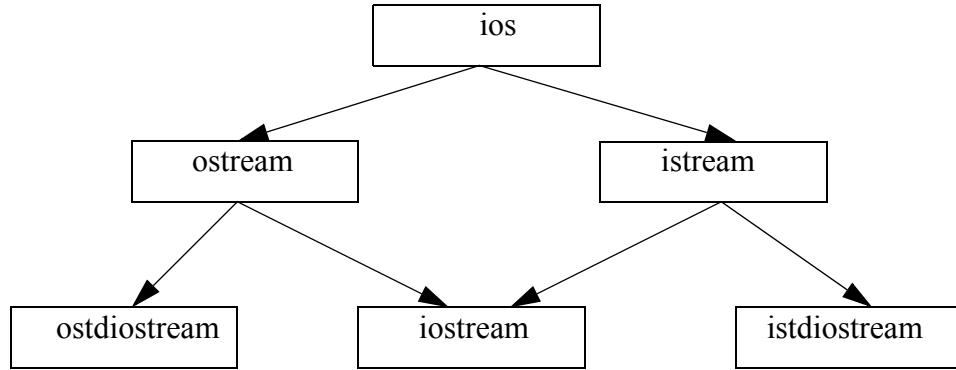
Headers

`iostream.h`

`streamb.h`

`stdiostr.h`

Classes



Example

```
#include <iostream.h>
void main() {
    // prints out "Hello World\n"
    cout << "Hello " << "World" << endl;
}
```

String Library

Description

This library component manages the handling of strings. While the ANSI-C library contains almost nothing to handle strings (see string.h), the C++ library provides a class String containing many operations on null-terminated sequences (strings), e.g. storage is allocated and freed as necessary by the member functions of the class String.

Headers

cstring.h

Classes

CString

Example

```
#include "cstring.h"

void main() {
    String s = "Mystring";
    s=s+": Hi!";
    printf(s); // prints out "Mystring: Hi!"
    if(s!="Mystring: Hi!") {
        // an error occurred !
    }
}
```

Memory Library

Description

This library contains the implementation of the operators “new” and “delete”. Those operators are needed to allocate and free memory.

Headers

new.h

Classes

None

Example

```
#include "new.h"

void main() {
    int *p = new int; // calls operator new defined in the
    library
    delete p;
}
```

Complex Library

Description

This library encapsulates complex arithmetic, such as addition, subtraction, multiplication, division, negation and much more. It defines 3 classes, one for each of the three floating-point types of Standard C (float, double, long double).

Headers

complex.h

Classes

```
float_complex
double_complex
long_double_complex
```

Example

```
#include "complex.h"
void main() {
    float_complex fc(1,2);
    if(fc.real()!=1) { /* ... error ... */ }
    if(fc.imag()!=2) { /* ... error ... */ }
    float_complex fc2;
    fc2=fc;
}
```

Bitset Library

Description

This library deals with bit arrays. This is useful for working with flags.

Headers

bitset.h

Classes

BitSet

Example

[Listing 4.1](#) shows code that uses the BitSet class.

Listing 4.1

```
#include <bitset.h>

void main() {
    BitSet bs, bs2;
    bs.set(4);
    bs2.set(3);
    bs=bs|bs2;
    if(bs.test(1)!=0) {
        /* Error occurred */
    }
}
```

```
    }
    if(bs.test(2)!=0) {
        /* Error occurred */
    }
    if(bs.test(3)==0) {
        /* Error occurred */
    }
    if(bs.test(4)==0) {
        /* Error occurred */
    }
    if(bs.test(5)!=0) {
        /* Error occurred */
    }
}
```

Strstream Library

Description

This library adapts the mechanism of streams to strings. Instead of having I/O channels, string streams work on a data buffer.

Headers

strstrea.h

Classes

ostrstream

istrstream

strstream

Example: #include <strstrea.h>

```
void main(void) {
    ostrstream os;
    os << "Hello" << endl;
}
```

Porting Tips and FAQs

This appendix describes some FAQs, and provides tips on how to port your application from a different tool vendor or the syntax of EBNF.

Migration Hints

This section describes the differences between this compiler and the compilers of other vendors. It also provides information about porting sources and how to adapt them.

Porting from Cosmic

If your current application is written for Cosmic compilers, there are some special things to consider.

How to Get Started...

The best way is if you create a new project using the New Project Wizard (in CodeWarrior IDE: Menu File > New) or a project from a stationery. This will set up a project for you with all the default options and library files included. Then add the existing files used for Cosmic to the project (e.g. through drag & drop from the Windows Explorer or using in the CodeWarrior IDE the menu Project > Add Files. You need to make sure that the right memory model and CPU type is used as for the Cosmic project.

Cosmic Compatibility Mode Switch

The latest Metrowerks compiler offers a Cosmic compatibility mode switch [-Ccx](#). Enable this compiler option so the compiler accepts most Cosmic constructs.

Assembly Equates

For the Cosmic compiler, you need to define equates for the inline assembly with `equ`. If you want to use the equates/values in C as well, you need to define it with `#define`'s as well. For Metrowerks, you only need one version (the `#define` one) both for C/C++ and for inline assembly. The `equ` is not supported in normal C/C++ code.

```
#ifdef __MWERKS__  
#define CLKSR_B 0x00 /* Clock source */  
#else  
CLKSR_B : equ $00 ; Clock source  
#endif
```

Inline Assembly Identifiers

For Cosmic, you need to place an underscore ('_') in front of each identifier, where for Metrowerks you can use the same name both for C/C++ and inline assembly. Additionally, for Metrowerks for better type-safety you need to place a '@' in front of variables if you want to use the address of a variable. Using conditional blocks like

```
#ifdef __MWERKS__  
idx @myVariable,x  
jsr MyFunction  
#else  
idx _myVariable,x  
jsr _MyFunction  
#endif
```

may be really painful. Using a macros which deal with the cases below are the better way to deal with this:

```
#ifdef __MWERKS__  
#define USCRA(ident) @ ident  
#define USCR(ident) ident  
#else /* for COSMIC, add a _ (underscore) to each ident */  
#define USCRA(ident) _##ident
```

```
#define USCRA(ident) _##ident
#endif
so the source can use the macros:
ldx USCRA(myVariable),x
jsr USCR(MyFunction)
```

Section Pragmas

Cosmic is using the `#pragma section` syntax, where Metrowerks is using either `#pragma DATA_SEG` or `#pragma CONST_SEG`:

```
#ifdef __MWERKS__
#pragma CONST_SEG CONSTVECT_SEG
#else
#pragma section const {CONSTVECT}
#endif
```

or another example (for data section):

```
#ifdef __MWERKS__
#pragma DATA_SEG APPLDATA_SEG
#else
#pragma section {APPLDATA}
#endif
```

Do not forget to use the segments (in the example above `CONSTVECT_SEG` and `APPLDATA_SEG`) in the linker .prm file in the `PLACEMENT` block.

Inline Assembly Constants

Cosmic is using an assembly constant syntax, where Metrowerks is using the normal C constant syntax:

```
#ifdef __MWERKS__
and 0xF8
#else
and ##$F8
```

```
#endif
```

Inline Assembly and Index Calculation

Cosmic is using the + operator to calculate offsets into arrays. For CodeWarrior, you have to use the ':' instead:

```
#ifdef __MWERKS__
    ldx array:7
#else
    ldx array+7
#endif
```

Inline Assembly and Tabs

Cosmic allows you to use TAB characters to be used in normal C strings (surrounded by double quotes):

```
asm(" this string contains hiddent tabs! ");
```

As the Metrowerks compiler rejects hidden tab characters in C strings according to the ANSI standard, you need remove the tab characters from such strings.

Inline Assembly and Operators

Cosmic and Metrowerks inline assembly may not support the same amount/level of operators. But in most cases it is simply to rewrite/transform them:

```
#ifdef __MWERKS__
    ldx #(BOFFIE + WUPIE)      ; enable Interrupts
#else
    ldx #(BOFFIE | WUPIE)      ; enable Interrupts
#endif
# ifdef __MWERKS__
    lda    #(_TxBuf2+Data0)
    ldx    #(( _TxBuf2+Data0) / 256)
#else
```

```
lda    #(_TxBuf2+Data0) & $ff)
ldx    #(_TxBuf2+Data0) >> 8) & $ff)
#endif
```

@interrupt

Cosmic is using the @interrupt syntax, where Metrowerks is using interrupt syntax. In order to keep the source base portable, a macro can be used (e.g. in a main header file which chooses the correct syntax depending on the compiler used:

```
/* e.g. in a header file: */
#ifndef __MWERKS__
#define INTERRUPT interrupt
#else
#define INTERRUPT @interrupt
#endif
/* now for each @interrupt we use the INTERRUPT macro: */
void INTERRUPT myISRFункция(void) { ....
```

Inline Assembly and Conditional Blocks

In most cases, the Cosmic compatibility switch [-Ccx](#) will handle the #asm blocks used in Cosmic inline assembly code. However, if #asm is used with conditional blocks like #ifdef or #if, then the C/C++ parser may not accept it:

```
void foo(void) {
#asm
    nop
#if 1
#endasm
    foo();
#asm
#endif
    nop
```

```
#endasm  
}
```

In such a case, the `#asm` and `#endasm` need to be ported to `asm { and }` block markers:

```
void foo(void) {  
    asm {  
        nop  
        #if 1  
        }  
        foo();  
        asm {  
            #endif  
            nop  
        }  
    }  
}
```

Compiler Warnings

Check carefully the warnings produced by the compiler. The Cosmic compiler does not warn about many cases where your application code may contain a bug. Later on you can switch off the warnings if they are ok (e.g. using the option [-W2](#) or using the [`#pragma MESSAGE`](#) in your source code).

Linker .lcf File (Cosmic) and Linker .prm file (CodeWarrior)

Cosmic is using a .lcf File for the linker with a special syntax. CodeWarrior is using a linker parameter file with a .prm file extension. The syntax is not the same format, but most things are straight forward to port. For CodeWarrior, you need to declare the RAM/ROM areas in the SEGMENTS ... END block, and the sections are placed into the SEGMENTS in the PLACEMENT...END block.

Make sure that all your segments you declared in your application (through `#pragma DATA_SEG`, `#pragma CONST_SEG` and `#pragma CODE_SEG`) are used in the PLACEMENT block of the linker prm file.

Check the linker warnings/errors carefully. They may be an indicator of what you need to adjust/correct in your application. E.g. you may have allocated the vectors in the linker .prm file (using VECTOR or ADDRESS syntax) and having it allocated it as well in the application itself (e.g. with #pragma CONST_SEG or with @address syntax). Allocating objects twice is an error, so you need to resolve this (do it one or the other way).

Consult your map file produced by the linker if everything is correctly allocated.

Remember that the Metrowerks linker is a smart linker. This means that objects not used/referenced are not linked to the application. The Cosmic linker may link objects even if they are not used/referenced, but you still may need to have them linked to the applications for any reasons for the Metrowerks linker. In order to have objects linked to the application regardless if they are used or not, use the ENTRIES ... END block in the linker .prm file:

```
ENTRIES /* the following objects/variables need to be  
linked even if not referenced by the application */  
_vectab ApplHeader FlashEraseTable  
END
```

Allocation of Bit Fields

Allocation of bitfields is very compiler-dependent. Some compilers allocate the bits first from right (LSByte) to left (MSByte), and others allocate from left to right. Also, alignment and byte/word crossing of bitfields is not implemented consistently. You may:

- Check the different allocation strategies.
- Check if there is an option to change the allocation strategy in the compiler.
- Use the compiler defines to hold sources portable:

```
__BITFIELD_LSBIT_FIRST__  
__BITFIELD_MSBIT_FIRST__  
__BITFIELD_LSBYTE_FIRST__  
__BITFIELD_MSBYTE_FIRST__  
__BITFIELD_LSWORD_FIRST__  
__BITFIELD_MSWORD_FIRST__
```

```
__BITFIELD_TYPE_SIZE_REDUCTION__
__BITFIELD_NO_TYPE_SIZE_REDUCTION__
```

Type Sizes, Sign of Character

Carefully check the type sizes of the compilers. Some compilers implement the sizes for the standard types (char, short, int, long, float, double) in different ways, e.g. int is for some compilers 16bit and for other ones 32bit.

If necessary change the default type settings with the `-l` option.

The sign of plain char is also not consistent over different Compilers. If the software relies that char is signed or unsigned, either change all plain char types to the signed or unsigned ones or change the sign of char with the `-T` option.

@bool Qualifier

Some compiler vendors are providing special keyword `@bool` to specify that a function returns a boolean value:

```
@bool int foo(void);
```

Because this is not supported, you have to remove this `@bool` or to use a define like this:

```
#define _BOOL /*@bool*/
_BOOL int foo(void);
```

@tiny/@far Qualifier for Variables

Some compiler vendors are providing special keywords to place variables to absolute locations. Such absolute locations can be expressed in ANSI-C as constant pointers:

```
#ifdef __HIWARE__
#define REG_PTB (*(volatile char*)(0x01))
#else /* other compiler vendors using non-ANSI features */
    @tiny volatile char REG_PTB      @0x01; /* port B */
#endif
```

The compiler does not need the `@tiny` qualifier directly, the compiler is smart enough to take the right addressing mode depending on the address:

```
/* compiler uses the correct addressing mode */
volatile char REG_PTB @0x01;
```

Arrays with Unknown Size

Some compilers accept following not-ANSI compliant statement to declare an array with unknown size:

```
extern char buf[0];
```

However, the compiler will issue an error message for this because an object with size zero (even if declared as extern) is illegal. Use the legal versions

```
extern char buf[ ];
```

Missing Prototype

Many compilers are accepting a function call usage without a prototype. This compiler will issue a warning for this. However if the prototype of a function with open arguments is missing and this function is called with different arguments numbers, this is clearly an error:

```
printf("hello world!"); // compiler assumes 'void
printf(char*)'
// error, argument number mismatch!
printf("hello %s!", "world");
```

To avoid such programming bugs use the compiler option [-Wpd](#) and always include/ provide a prototype.

asm("sequence")

Some compilers are using `_asm("string")` to write inline assembly in normal C source: `_asm("nop");`

This can be rewritten with `asm` or `asm { };`: `asm nop;`

Recursive Comments

Some compilers accept recursive comments without any warnings. This compiler will issue a warning for each such recursive comment:

```
/* this is a recursive comment  /*
int a;
/* */
```

This compiler will treat the above source completely as one single comment, so the definition of ‘a’ is inside the comment, that is the compiler treats all between the first opening comment ‘/*’ until the closing comment token ‘*/’ as a comment. If there are such recursive comments, correct them.

Interrupt Function, @interrupt

Interrupt functions have to be marked with the #pragma TRAP_PROC or using the interrupt keyword (see Front End, Back End).

```
#ifdef __HIWARE__
    #pragma TRAP_PROC
    void MyTrapProc(void)
#else /* other compiler-vendor non-ANSI declaration of
       interrupt
               function */
    @interrupt void MyTrapProc(void)
#endif
{
    /* code follows here */
}
```

Defining Interrupt Functions

This manual entry discusses some important topics related to the handling of interrupt functions:

- Definition of an interrupt function
- Initialization of the vector table
- Placing interrupt function in special sections

Defining an Interrupt Function

The compiler provides two ways to define an interrupt function:

- Using pragma TRAP_PROC.
- Using the keyword interrupt.

Using Pragma “TRAP_PROC”

The pragma TRAP_PROC indicates to the compiler that the following function is an interrupt function. In that case, the compiler should terminate the function by a special interrupt return sequence (for many processors a RTI instead of a RTS).

Example:

```
#pragma TRAP_PROC  
void INCcount(void) {  
    tcount++;  
}
```

Using Keyword “interrupt”

The keyword ‘interrupt’ is not ANSI C standard and as thus is not supported by all ANSI C compiler vendor. In the same way, the syntax for the usage of this keyword may change between different compiler. The keyword interrupt indicates to the compiler that the following function is an interrupt function.

Example:

```
interrupt void INCcount(void) {  
    tcount++;  
}
```

Initializing the Vector Table

Once the code for an interrupt function has been written, you have to associate this function with an interrupt vector. This is done through initialization of the vector table. The vector table can be initialized in following ways:

- Using the command VECTOR or VECTOR ADDRESS in the PRM file
- Using the keyword “interrupt”.

Using the Linker Commands

The Linker provides two commands to initialize the vector table: VECTOR or VECTOR ADDRESS. The command VECTOR ADDRESS is used to write the address of a function at a specific address in the vector table.

Example:

In order to enter the address of the function INCcount at address 0x8A, insert following command in the application PRM file:

```
VECTOR ADDRESS 0x8A INCcount
```

The command VECTOR is used to associate a function with a specific vector, identified with its number. The mapping from the vector number is target specific.

Example:

In order to associate the address of the function INCcount with the vector number 69, insert following command in the application PRM file:

```
VECTOR ADDRESS 69 INCcount
```

Using the Keyword “interrupt”

When you are using the keyword “interrupt”, you may directly associate your interrupt function with a vector number in the ANSI C source file. In that purpose just specify the vector number next to the keyword interrupt.

Example:

In order to associate the address of the function INCcount with the vector number 69, define your function as follows:

```
interrupt 69 void INCcount(void) {
    int card1;
    tcount++;
}
```

Placing Interrupt Function in Special Sections

For all targets supporting paging, the interrupt function must be allocated in an area which is accessible all the time. This can be obtained by putting the interrupt function into a specific segment.

Defining Functions in a Specific Segment.

In order to define a function in a specific segment, you can use the pragma CODE_SEG.

Example:

```
/* These functions are defined in segment 'int_Function' */
#pragma CODE_SEG Int_Function
#pragma TRAP_PROC

void INCcount(void) {
    tcount++;
}

#pragma CODE_SEG DEFAULT /*Return to default code segment. */
```

Placing a Specific Segment.

In the PRM file, you can define where you want to allocate each segments you have defined in your source code. In order to place a segment in a specific memory area; just add the segment name in the PLACEMENT block of your PRM file. Be careful, the linker is case sensitive. Take special attention to the upper and lower cases in your segment name.

Example:

```
LINK test.abs

NAMES test.o ... END

SECTIONS
    INTERRUPT_ROM = READ_ONLY      0x4000 TO 0x5FFF;
    MY_RAM        = READ_WRITE     .....

PLACEMENT
    Int_Function           INTO INTERRUPT_ROM;
    DEFAULT_RAM            INTO MY_RAM;
    ....
END
```

How to Use Variables in EEPROM

Placing variables into EEPROM is not explicitly supported in the C/C++ language, but because EEPROM is widely available in embedded processors, a development tools for Embedded Systems has to support it.

The examples are processor-specific. However, it is very easy to adapt them for any other processor.

Linker Parameter File

You have to define your RAM/ROM areas in your linker parameter file. However, you should declare the EEROM memory as NO_INIT to avoid the memory range being initialized during normal startup.

```
LINK test.abs

NAMES test.o startup.o ansi.lib END
SECTIONS
```

```
MY_RAM = READ_WRITE 0x800 TO 0x801;
MY_ROM = READ_ONLY   0x810 TO 0xAFF;
MY_STK = READ_WRITE 0xB00 TO 0xBFF;
EEPROM = NO_INIT     0xD00 TO 0xD01;

PLACEMENT
DEFAULT_ROM    INTO MY_ROM;
DEFAULT_RAM    INTO MY_RAM;
SSTACK          INTO MY_STK;
EEPROM_DATA    INTO EEPROM;

END
/* set reset vector to function _Startup defined in startup
code */
VECTOR ADDRESS 0xFFFFE _Startup
```

The Application

The example in [Listing A.1](#) shows an application example erasing or writing a EEPROM word. The example is specific to the processor used, but if you consult your technical documentation about the EEPROM used for your derivative or CPU, it is easy to adapt.

NOTE	There are only a limited number of write operations guaranteed for EEPROMs, so avoid writing to a EEPROM cell to frequently.
-------------	------------------------------------------------------------------------------------------------------------------------------

Listing A.1 Erasing and Writing an EEPROM

```
/*
Definition of a variable in EEPROM.

The variable VAR is located in EEPROM.
- It is defined in a user defined segment EEPROM_DATA
- In the PRM file EEPROM_DATA is placed at address 0xD00.
Be careful, The EEPROM can only be written a limited number of time.
Running this application a long time may reach this limit and your
EEPROM may be unusable afterward.
*/
#include <hidef.h>
#include <stdio.h>
#include <math.h>
/* INIT register. */
typedef struct {
    union {
```

```
struct {
    unsigned int    bit0:1;
    unsigned int    bit1:1;
    unsigned int    bit2:1;
    unsigned int    bit3:1;
    unsigned int    bit4:1;
    unsigned int    bit5:1;
    unsigned int    bit6:1;
    unsigned int    bit7:1;
} INITEE_Bits;
unsigned char INITEE_Byt;
} INITEE;
} INIT;
volatile INIT INITEE @0x0012;
#define EEON INITEE.INITEE.INITEE_Bits.bit0
/* EEPROM register. */
volatile struct {
    unsigned int    EEPGM:1;
    unsigned int    EELAT:1;
    unsigned int    ERASE:1;
    unsigned int    ROW:1;
    unsigned int    BYTE:1;
    unsigned int    dummy1:1;
    unsigned int    dummy2:1;
    unsigned int    BULKP:1;
} EEPROM @0x00F3;
/* EEPROT register. */
volatile struct {
    unsigned int    BPROT0:1;
    unsigned int    BPROT1:1;
    unsigned int    BPROT2:1;
    unsigned int    BPROT3:1;
    unsigned int    BPROT4:1;
    unsigned int    dummy1:1;
    unsigned int    dummy2:1;
    unsigned int    dummy3:1;
} EEPROT @0x00F1;
#pragma DATA_SEG EEPROM_DATA
unsigned int VAR;
#pragma DATA_SEG DEFAULT
void EraseEEPROM(void) {
    /* Function used to erase one word in the EEPROM. */
    unsigned long int i;
    EEPROM.BYTE =1;
    EEPROM.ERASE =1;
```

Porting Tips and FAQs

How to Use Variables in EEPROM

```
EEPROM.EELAT =1;
VAR = 0;
EEPROM.EEPGM =1;
for (i = 0; i<4000; i++) {
/* Wait until EEPROM is erased. */
}
EEPROM.EEPGM =0;
EEPROM.EELAT =0;
EEPROM.ERASE =0;
}

void WriteEEPROM(unsigned int val) {
/* Function used to write one word in the EEPROM. */
unsigned long int i;
EraseEEPROM();
EEPROM.ERASE =0;
EEPROM.EELAT =1;
VAR = val;
EEPROM.EEPGM =1;
for (i = 0; i<4000; i++) {
/* Wait until EEPROM is written. */
}
EEPROM.EEPGM =0;
EEPROM.EELAT =0;

}

void func1(void) {
unsigned int i;
unsigned long int ll;
i = 0;
do
{
    i++;
    WriteEEPROM(i);
    for (ll = 0; ll<200000; ll++) {
    }
}
while (1);

}

void main(void) {
EEPROM.BPROT4 = 0;
EEON=1;
```

```
WriteEEPROM( 0 );
func1();
}
```

General Optimization Hints

Here are some hints how to reduce your application size:

- Check if you need the full startup code: e.g. if you do not have initialized data, you can ignore/remove the copy-down, if you do not need to have the memory initialized, you can remove the zero-out. And if you do not need both, you may remove the complete startup code, and set up your stack in your main routine directly. Set with `INIT main` in the prm file your main routine as application startup/entry
- Check the compiler options: you may use e.g. the [-OdocF](#) function: increases compilation speed, but decreases code size. E.g. you can try `-OdocF="-or"`. Using the option [-Ll](#) to write a log file tells you the effect for each single function.
- Check if you can use both IEEE32 for float and double. See the option [-T](#) how to configure this. Do not forget to link the corresponding ANSI/C++ library.
- Use smaller data types whenever possible (e.g. 16bit instead of 32bit).
- Have a look into the map file check runtime routines which usually have a `'_'` prefix. Check for 32bit integral routines (e.g. `_LADD`). Check if you need the long arithmetic.
- Enumerations: if you are using enums, by default they have the size of 'int'. They can be set to an unsigned 8bit (see option [-T](#), or use `-TE1uE`).
- Check if you are using switch tables (have a look into the map file as well). there are options to configure this (see [-CswMinSLB](#) as an example)
- Finally the linker has an option to overlap ROM areas (see `-COCC` option in the linker)

Executing an Application from RAM

For performance reason, it may be interesting to copy an application from ROM to RAM, and to execute it from RAM. This can be achieved following the procedure below.

1. Link your application with code located in RAM.
2. Generate a S Record File.

-
3. Modify the startup code to copy the application code.
 4. Link the application with the S record file previously generated.

Each step is described in the following sections. As an example, we will take the application `fibo.abs`.

Link your application with code located in RAM.

We recommend that you generate a ROM library for your application. This allows you to easily debug your final application (including the copying of the code).

ROM Library Startup File

A ROM Library requires a very simple startup file, containing only the definition from the startup structure. Usually a ROM library startup file looks as follows:

```
#include "startup.h"

/* read-only: _startupData is allocated in ROM and ROM
Library PRM File */

struct _tagStartup _startupData;
```

You must generate a PRM file, where the code is placed in RAM. As the compiler generates absolute code, the linker should know the final location of the code in order to generate correct code for the function call.

Additionally, in the PRM file specify the name of the application entry points in the ENTRIES block. The application main function, as well as the function associated with an Interrupt vector, must be specified there.

Example

Suppose you want to copy and execute your code at address 0x7000. Your PRM file will look as follows:

```
LINK fiboram.abs AS ROM_LIB

NAMES myFibo.o start.o END
SECTIONS
    MY_RAM = READ_WRITE 0x4000 TO 0x43FF;
    MY_ROM = READ_ONLY   0x7000 TO 0xBFFF; /* Destination
Address in RAM area */
PLACEMENT
    DEFAULT_ROM, ROM_VAR, STRINGS INTO MY_ROM;
    DEFAULT_RAM           INTO MY_RAM;
END
```

```
ENTRIES
    myMain
END
```

NOTE

You cannot use a function `main` in a ROM library. Please use another name for the application entry point. In the example above, we have used "myMain".

Generate an S Record File

An S record must be generated for the application. In this purpose, the Burner utility can be used.

The file is generated when you click the '1st byte(msb)' button in the burner dialog.

Note: The field 'From' must be initialized with 0 and the field length with a value bigger than the last byte used for the code (if byte `0xFFFF` is used, the Length must be at least `10000`).

Modify the startup code

The startup code of the final application must be modified. It should contain code copying the code from RAM to ROM. Additionally, as the application entry point is located in the ROM library, it should call it explicitly.

Application PRM file

The S record file (generated previously) must be linked to the application with an offset.

Example

Suppose the application code must be placed at address `0x800` in ROM and should be copied to address `0x7000` in RAM. The application PRM file looks as follows:

```
LINK fiboram.abs

NAMES mystart.o fiboram.abs ansis.lib END
SECTIONS
    MY_RAM = READ_WRITE 0x5000 TO 0x53FF;
    MY_ROM = READ_ONLY   0x0600 TO 0x07FF;
```

```
PLACEMENT
    DEFAULT_ROM, ROM_VAR, STRINGS INTO MY_ROM;
    DEFAULT_RAM                 INTO MY_RAM;
END
STACKSIZE 0x100
VECTOR 0 _Startup /* set reset vector on startup function */
HEXFILE fiboram.s1 OFFSET 0xFFFFF9800 /* 0x800 - 0x7000 */
```

Note: The offset specified in the HEXFILE command is added to each record in the S record file. The code at address 0x700 will be encoded at address 0x800.

If you are using CodeWarrior, then the CodeWarrior IDE will pass all the names in the NAMES...END part directly to the linker, so then the NAMES...END part will be empty.

Copying Code from ROM to RAM

A function must be implemented, which copy the code from ROM to RAM.

Example

Suppose the application code must be placed at address 0x800 in ROM and should be copied to address 0x7000 in RAM the copy function can be implemented as follows:

```
/* Start address of the application code in ROM. */
#define CODE_SRC 0x800

/* Destination address of the application code in RAM. */
#define CODE_DEST 0x7000

#define CODE_SIZE 0x90 /* Size of the code which must be
copied.*/

void CopyCode(void) {
    unsigned char *ptrSrc, *ptrDest;

    ptrSrc = (unsigned char *)CODE_SRC;
    ptrDest = (unsigned char *)CODE_DEST;
    memcpy (ptrDest, ptrSrc, CODE_SIZE);
}
```

Invoking the Application Entry Point in the Startup Function

The startup code should call the application entry point, which is located in the ROM library. This function must be explicitly called through its name. The best place is just before calling the application main routine.

Example

```
void _Startup(void) {
    ... set up stack pointer ...
    ... zero out ...
    ... copy down ...
    CopyCode();
    ... call main ...
}
```

Defining a Dummy Main Function

The linker cannot link an application if there is no main function available. As in our case, the main function is located in the ROM library; a dummy main function must be defined in the startup module.

Example

```
#pragma NO_ENTRY
#pragma NO_EXIT
void main(void) {
    asm NOP;
}
```

Frequently Asked Questions (FAQs), Trouble Shooting

This section provides some tips on how to solve the most commonly encountered problems.

Making Applications

If the compiler or linker crashes, isolate the construct causing the crash and send us a bug report. Other common problems are:

The compiler reports an error, but WinEdit doesn't display it.

This means that WinEdit didn't find the EDOOUT file, i.e. the compiler wrote it to a place not expected by WinEdit. This can have several reasons. Check that the environment variable [DEFAULTDIR](#) is not set, and that the project directory is set correctly. Also in WinEdit 2.1, make sure that the OUTPUT entry in the file WINEDIT.INI is empty.

Some programs can't find a file.

Make sure your environment is set up correctly. Also check WinEdit's project directory. Read the [Input Files](#) section of this document.

The compiler seems to generate incorrect code.

First make sure the code is incorrect. Sometimes the operator-precedence rules of ANSI-C do not quite give the results one would expect. Apparently faulty code can appear to be correct. Consider the following example:

```
if (x & y != 0) ...
```

is evaluated as

```
if (x & (y != 0)) ...
```

not as:

```
if ((x & y) != 0) ...
```

Another source of unexpected behavior are the integral promotion rules of C, that is to say, that characters are usually (sign-)extended to integers. This can sometimes have quite unexpected effects, e.g. the if-condition below is FALSE:

```
unsigned char a, b;  
b = -8;  
a = ~b;  
if (a == ~b) ...
```

because extending a results in 0x0007, while extending b gives 0x00F8 and the '~' results in 0xFF07. If the code contains a bug, isolate the construct causing it and send us a bug report.

The code seems to be correct, but the application does not work.

Check if the hardware is not set up correctly (e.g. using chip selects). Some memory expansions are accessible only with a special access mode (e.g. only word accesses). If memory is accessible only in a certain way, use inline assembly or use the ‘volatile’ keyword.

The linker can't handle an object file

Make sure all object files have been compiled with the latest version of the compiler and with the same flags concerning memory models and floating point formats. If not, recompile them.

The make utility doesn't make all of the application.

Most probably you didn't specify the target that is to be made on the command line. In this case, the make utility assumes the target of the first rule is the top target. Either put the rule for your application as the first in the make file, or specify the target on the command line.

The make utility unnecessarily re-compiles a file.

This problem can appear if you have short source files in your application. It is caused by the fact that MS-DOS only saves the time of last modification of a file with an accuracy of ± 2 seconds. If the compiler compiles two files in that time, both will have the same time stamp. The make utility makes the safe assumption that if one file depends on another file with the same time stamp, the first file has to be recompiled. There's no way to solve this problem.

The help file cannot be opened by double clicking on it in the file manager or in the explorer.

The compiler help file is a true Win32 help file. It is not compatible with the windows 3.1 version of WinHelp. The program “winhelp.exe” delivered with Windows 3.1, Windows 95 and Windows NT can only open Windows 3.1 help files. To open the compiler help file, the Winhlp32.exe has to be used.

The program winhlp32.exe resides either in the windows directory (usually c:\windows, c:\win95 or c:\winnt) or in its system (Win32s) or system32 (Windows

95, WinNT) subdirectory. The Winhlp32.exe is also contained in the Win32s distribution.

To change the association with Windows 95 or Windows NT either use the explorer menu "View->Options" and there the "File Types" tab. Or select any help file, press the Shift key. Hold it while opening the context menu by clicking on the right mouse button. Select "Open with ..." from the menu. Enable the "Always using this program" check box and select the winhlp32.exe with the "other" button.

To change the association with the file manager under Windows 3.1 use the "File->Associate..." menu entry.

How can constant objects be allocated in ROM?

Use the [#pragma INTO_ROM](#) and see the compiler option [-Cc](#).

The compiler cannot find my source file. What is wrong?

Check if in the default.env file the path to the source file is set in the environment variable [GENPATH](#). Additionally you can use the compiler option [-I](#) to specify the include file path. With CodeWarrior check the access path in the preference panel.

How can I switch off smart linking?

By adding a '+' after the object in the NAMES list of the prm file.

With CodeWarrior and ELF/DWARF object file format (see compiler option [-F](#)), Today, you can link all in the object within an ENTRIES...END directive in the linker prm file:

```
ENTRIES fibo.o: * END
```

This is NOT supported in the HIWARE object file format.

How to avoid announcing 'no access to memory'?

Change in the simulator/debugger the memory configuration mode (menu Simulator > Configure) to 'auto on access'.

How can the same memory configuration be loaded every time the simulator/debugger is started?

Save that memory configuration under default.mem ex: Simulator->Configure-> Save and type in default.mem.

How can a loaded program in the simulator/debugger be started automatically and stop at a specified breakpoint?

Define the file postload.cmd, example:

```
bs &main t  
g
```

How can a project file be set up with WinEdit?

Open Win Edit press Project->Configure->Open and load a *.wpj file.

How can a project file be set up with Codewright?

Open Codewright press Project->New and load a *.pj file or create one.

Where can an overview of all options for the compiler be found?

Type in [H](#) at the command line of the compiler.

How can an own startup be called after reset?

In the prm file use:

```
INIT myStartup
```

How can an own name for main be used?

In the prm file use:

```
MAIN myMain
```

How can the reset vector be set to the beginning of the startup code?

Set in the prm file for example this line:

```
/* set reset vector on _Startup */
VECTOR ADDRESS 0xFFFFE _Startup
```

How can the compiler be configured for the editor?

Open the compiler press File->Configuration choose Editor Settings.

Where are configuration settings saved?

In the file project.ini. With CodeWarrior, the compiler settings are stored in the .mcp file.

What is to do when 'error while adding default.env options' appears after starting the compiler?

Adapt the options set by the compiler to the ones set in the default.env file and save them by pressing the save button in the compiler into project.ini.

After starting up the ICD Debugger an error "Illegal breakpoint detected" appears. What is wrong?

The cable might be too long, the maximal length for unshielded cables is about 20 cm and depends on the electric noise in the environment.

Why can no initialized data be written into the ROM area?

The qualifier const must be used and the source must be compiled with option [-Cc](#).

Problems in the communication, loosing communication.

The cable might be too long, the maximal length for unshielded cables is about 20 cm and depends on the electric noise in the environment.

What to do if an assertion happens (internal error)?

Extract source where assertion appears and send it as zipped file with all headers, used options and versions of all used tools.

How to get help on an error message?

Either press F1 after clicking on the message to start up the help file or copy the message number, open the pdf manual and make a search on the copied message number.

How to get help on an option?

Open a compiler and type into the command line [H](#), a list of all options appears with a short description of it, or look into the manual for detailed information. Another way is to press F1 in the options setting dialog while a option is marked.

Bug Reports

If you cannot solve your problem, you may need to contact our Technical Support Department. Isolate the problem – if it's a compiler problem, write a short program reproducing the problem. Then send us a bug report.

Send or fax your bug report to your local distributor, it will be forwarded to the Technical Support Department.

The report type gives us a clue how urgent a bug report is. The classification is:

Information

This section describes things you would like to see improved in a future major release.

Bug

An error for which you have a workaround or would be satisfied for the time being if we could supply a workaround. (If you already have a workaround, we'd like to know it, too!) Of course bugs will be fixed in the next release.

Critical Bug

A grave error that makes it impossible for you to continue with your work.

Electronic Mail (email) or Fax Report Form

If you send the report by fax or email, following template can be used.

REPORT FORM

(Fill this form and send it using:
EMail: support_europe@metrowerks.com
Fax : ++(41) 61 690 75 05
)

CUSTOMER INFORMATION

Customer Name :
Company :
Customer Number:
Phone Number:
Fax Number:
Email Address:

PRODUCT INFORMATION

Product:
Host Computer (PC, ...):
OS/Window Manager (WinNT, Win95, ...):
Target Processor:
Language (C, C++, ...):

TOOL INFORMATION

Tool (Compiler, Linker, ...):
Version Nr (Vx.x.xx):
Options Used:
For simulator/debugger only: Target Interface Used:

REPORT INFORMATION

Report Type (Bug, Wish, Information):
Severity Level (0: Higher, ... 5: Lower):
(0 : No workaround, development stopped.
1 : Workaround found, can continue development, problems seems to
be a common one.
2 : Workaround found, problem with very special code.

```
3 : Has to be improved.  
4 : Wish  
5 : Information  
)  
Description:  
Source/Preprocessor output:
```

Technical Support

The best way to get technical support is by using electronic mail (email). It is also possible to attach some examples to the email using a compression utility (e.g. WinZip) or simply uuencode. The email address is:

`support_europe@metrowerks.com`

To get information about newest updates and product enhancements, visit the web page at:

`http://www.metrowerks.com`

To reach technical support by postal mail, use the address below:

Metrowerks AG
Technical Support
Riehenring 175
4058 Basel (Switzerland)
Phone: ++41 61 690 7505
Fax: ++41 61 690 7501
Email: `support_europe@metrowerks.com`

Metrowerks
7700 West Parmer Lane
Austin, TX 78729 USA
Email: `support@metrowerks.com`

EBNF Notation

This chapter gives a short overview of the Extended Backus–Naur Form (EBNF) notation, which is frequently used in this document to describe file formats and syntax rules. A short introduction to EBNF is presented.

EBNF Example

```
ProcDecl      = PROCEDURE "(" ArgList ")" .  
ArgList       = Expression { "," Expression} .  
Expression    = Term ("*" | "/") Term .  
Term          = Factor AddOp Factor .  
AddOp         = "+" | "-" .  
Factor        = ([ "-"] Number) | "(" Expression ")" .
```

The EBNF language is a formalism that can be used to express the syntax of context-free languages. The EBNF grammar consists of a rule set called – *productions* of the form:

```
LeftHandSide = RightHandSide .
```

The left-hand side is a non-terminal symbol. The right-hand side describes how it is composed.

EBNF consists of the symbols discussed in the sections that follow.

Terminal Symbols

Terminal symbols (terminals for short) are the basic symbols which form the language described. In above example, the word **PROCEDURE** is a terminal. Punctuation symbols of the language described (not of EBNF itself) are quoted (they are terminals, too), while other terminal symbols are printed in **boldface**.

Non-Terminal Symbols

Non-terminal symbols (non-terminals) are syntactic variables and have to be defined in a production, i.e. they have to appear on the left hand side of a production somewhere. In above example, there are many non-terminals, e.g. **ArgList** or **AddOp**.

Vertical Bar

The vertical bar " | " denotes an alternative, i.e. either the left or the right side of the bar can appear in the language described, but one of them has to. E.g. the 3rd production above means “an expression is a term followed by either a “*” or a “/” followed by another term”.

Brackets

Parts of an EBNF production enclosed by "[" and "] " are optional. They may appear exactly once in the language, or they may be skipped. The minus sign in the last production above is optional, both -7 and 7 are allowed.

The repetition is another useful construct. Any part of a production enclosed by " { " and " } " may appear any number of times in the language described (including zero, i.e. it may also be skipped). ArgList above is an example: an argument list is a single expression or a list of any number of expressions separated by commas. (Note that the syntax in the example does not allow empty argument lists...)

Parentheses

For better readability, normal parentheses may be used for grouping EBNF expressions, as is done in the last production of the example. Note the difference between the first and the second left bracket. The first one is part of the EBNF notation. The second one is a terminal symbol (it is quoted) and may appear in the language.

Production End

A production is always terminated by a period.

EBNF-Syntax

The definition of EBNF in EBNF is:

```
Production  = NonTerminal "=" Expression "..".
Expression   = Term {"|" Term}.
Term         = Factor {Factor}.
Factor       = NonTerminal
              | Terminal
              | "(" Expression ")"
              | "[" Expression "]"
              | "{" Expression "}".
Terminal     = Identifier | """ <any char> """.
NonTerminal  = Identifier.
```

The identifier for a non-terminal can be any name you like. Terminal symbols are either identifiers appearing in the language described or any character sequence that is quoted.

Extensions

In addition to this standard definition of EBNF, the following notational conventions are used.

The counting repetition: Anything enclosed by "`{`" and "`}`" and followed by a superscripted expression x must appear exactly x times. x may also be a non-terminal. In the following example, exactly four stars are allowed:

$$\text{Stars} = \{ "*" \}^4.$$

The size in bytes. Any identifier immediately followed by a number n in square brackets ("[" and "]") may be assumed to be a binary number with the most significant byte stored first, having exactly n bytes. Example:

$$\text{Struct} = \text{RefNo} \text{ FilePos}[4].$$

In some examples, text is enclosed by "`<`" and "`>`". This text is a meta-literal, i.e. whatever the text says may be inserted in place of the text. (cf. `<any char>` in the above example, where any character can be inserted).

Abbreviations, Lexical Conventions

Topic	Description
ANSI	American National Standards Institute
Compilation Unit	Source file to be compiled, includes all included header files
Floating Type	Numerical type with a fractional part, e.g. float, double, long double
HLI	High Level Inline Assembly
Integral Type	Numerical type without fractional part, e.g. char, short, int, long, long long

Number Formats

Valid constant floating number suffixes are '`f`' and '`F`' for float and '`l`' or '`L`' for long double. Note that floating constants without suffixes are double constants in ANSI. For exponential numbers '`e`' or '`E`' has to be used. '`-`' and '`+`' can be used for signed representation of the floating number or the exponent.

Following suffixes are supported:

Constant	Suffix	Type
floating	F	float
floating	L	long double
integral	U	unsigned int
integral	uL	unsigned long

Suffixes are not case-sensitive, e.g. ‘ul’, ‘Ul’, ‘uL’ and ‘UL’ all denote a unsigned long type.

Examples:

```
+3.15f    /* float */
-.125f   /* float */
3.125f   /* float */
.787F    /* float */
7.125    /* double */
3.E7     /* double */
8.E+7    /* double */
9.E-7    /* double */
3.2l     /* long double */
3.2e12L  /* long double */
```

Precedence and Associativity of Operators for ANSI C

The following table gives an overview of the precedence and associativity of operators.

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right

Porting Tips and FAQs

Precedence and Associativity of Operators for ANSI C

Operators	Associativity
<code>== !=</code>	left to right
<code>&</code>	left to right
<code>^</code>	left to right
<code> </code>	left to right
<code>&&</code>	left to right
<code> </code>	left to right
<code>? :</code>	right to left
<code>= += -= *= /= %= &= ^= = <<= >>=</code>	right to left
<code>,</code>	left to right

NOTE Unary `+`, `-` and `*` have higher precedence than the binary forms.

The precedence and associativity is determined by the ANSI-C syntax (ANSI/ISO 9899-1990, p. 38 and Kernighan/ Ritchie, “*The C Programming Language*”, Second Edition, Appendix Table 2-1).

Examples:

```
if(a == b&&c)
```

is the same as

```
if ((a ==b) && c)
```

but

```
if (a == b|c)
```

is the same as

```
if ((a == b) | c)
```

```
a = b + c * d;
```

In the example above, operator-precedence causes the product of `(c*d)` to be added to `b`, and that sum to be assigned to `a`.

```
a = b += c += 1;
```

The associativity rules first evaluate `c+=1`, then assign `b` to the value of `b` plus `(c+=1)`, and then the result to `a`.

List of all Escape Sequences

The following table gives an overview over escape sequences which could be used inside strings (e.g. for printf):

Description	Escape Sequence
Line Feed	\n
Tabulator sign	\t
Vertical Tabulator	\v
Backspace	\b
Carriage Return	\r
Line feed	\f
Bell	\a
Backslash	\\\
Question Mark	\?
Quotation Mark	\^
Double Quotation Mark	\"
Octal Number	\ooo
Hexadecimal Number	\xhh

Porting Tips and FAQs

List of all Escape Sequences

Global Configuration File Entries

This appendix documents the entries that can appear in the global configuration file. This file is named MCUTOOL.INI.

MCUTOOLS.INI can contain these sections:

- [\[Options\] Section](#)
- [\[XXX_Compiler\] Section](#)
- [\[Editor\] Section](#)
- [Example](#)

[Options] Section

This section documents the entries that can appear in the [Options] section of the file MCUTOOLS.INI.

DefaultDir

Arguments

Default Directory to be used.

Description

Specifies the current directory for all tools on a global level (see also environment variable [DEFAULTDIR](#)).

Example

```
DefaultDir=c:\install\project
```

[XXX_Compiler] Section

This section documents the entries that can appear in an [xxx_Compiler] section of the file MCUTOOLS.INI.

NOTE XXX is a placeholder for the name of the actual backend. For example, for the HC12 compiler, the name of this section would be [HC12_Compiler].

SaveOnExit

Arguments

1/0

Description

Set to 1 if the configuration should be stored when the compiler is closed. Set to 0 if it should not be stored. The compiler does not ask to store a configuration in either case.

SaveAppearance

Arguments

1/0

Description

Set to 1 if the visible topics should be stored when writing a project file. Set to 0 if not. The command line, its history, the windows position, and other topics belong to this entry.

SaveEditor

Arguments

1/0

Description

Set to 1 if the visible topics should be stored when writing a project file. Set to 0 if not. The editor setting contains all information of the editor configuration dialog.

SaveOptions

Arguments

1/0

Description

Set to 1 if the options should be saved when writing a project file. Set to 0 if the options should not be saved. The options also contain the message settings.

RecentProject0, RecentProject1, ...

Arguments

Names of the last and prior project files

Description

This list is updated when a project is loaded or saved. Its current content is shown in the file menu.

Global Configuration File Entries

[XXX_Compiler] Section

Example

```
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=C:\myprj\project.ini
RecentProject1=C:\otherprj\project.ini
```

TipFilePos

Arguments

Any integer, e.g. 236

Description

Actual position in tip of the day file. Used that different tips are shown at different calls.

Saved

Always saved when saving a configuration file.

ShowTipOfDay

Arguments

0/1

Description

Should the Tip of the Day dialog be shown at startup.

1: it should be shown

0: Only when opened in the help menu

Saved

Always saved when saving a configuration file.

TipTimeStamp

Arguments

date and time

Description

Date and time when the tips were last used.

Saved

Always saved when saving a configuration file.

[Editor] Section

This section documents the entries that can appear in the [Editor] section of the file MCUTOOLS.INI.

Editor_Name

Arguments

The name of the global editor

Description

Specifies the name which is displayed for the global editor. This entry has only a description effect. Its content is not used to start the editor.

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Global Configuration File Entries

[Editor] Section

Editor_Exe

Arguments

The name of the executable file of the global editor

Description

Specifies the file name that is called (for showing a text file) when the global editor setting is active. In the editor configuration dialog, the global editor selection is active only when this entry is present and not empty.

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Editor_Opts

Arguments

The options to use the global editor

Description

Specifies options used for the global editor. If this entry is not present or empty, “%f” is used. The command line to launch the editor is built by taking the Editor_Exe content, then appending a space followed by this entry.

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Example

```
[Editor]
editor_name=notepad
editor_exe=C:\windows\notepad.exe
editor_opts=%f
```

Example

[Listing B.1](#) shows a typical MCUTOOLS.INI file.

Listing B.1 A Typical MCUTOOLS.INI File Layout

```
[Installation]
Path=c:\Metrowerks
Group=ANSI-C Compiler

[Editor]
editor_name=notepad
editor_exe=C:\windows\notepad.exe
editor_opts=%f

[Options]
DefaultDir=c:\myprj

[XXXX_Compiler]
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=c:\myprj\project.ini
RecentProject1=c:\otherprj\project.ini
TipFilePos=0
ShowTipOfDay=1
TipTimeStamp=Jan 21 2000 17:25:16
```

Global Configuration File Entries

Example

Local Configuration File Entries

This appendix documents the entries that can appear in the local configuration file. Usually, you name this file *project.ini*, where *project* is a placeholder for the name of your project.

A *project.ini* file can contains these sections:

- [\[Editor\] Section](#)
- [\[XXX_Compiler\] Section](#)
- [Example](#)

[Editor] Section

Editor_Name

Arguments

The name of the local editor

Description

Specifies the name that is displayed for the local editor. This entry contains only a description effect. Its content is not used to start the editor.

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog. This entry has the same format as the global editor configuration in the `mcutools.ini` file.

Local Configuration File Entries

[Editor] Section

Editor_Exe

Arguments

The name of the executable file of the local editor

Description

Specifies the file name that is used for a text file, when the local editor setting is active. In the editor configuration dialog, the local editor selection is only active when this entry is present and not empty.

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog. This entry has the same format as for the global editor configuration in the mcutools.ini file.

Editor_Opts

Arguments

Local editor options

Description

Specifies options that should be used for the local editor. If this entry is not present or empty, "%f" is used. The command line to launch the editor is built by taking the Editor_Exe content, then appending a space followed by this entry.

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog. This entry has the same format as the global editor configuration in the mcutools.ini file.

Example [Editor] Section

```
[Editor]
editor_name=notepad
editor_exe=C:\windows\notepad.exe
editor_opts=%f
```

[XXX_Compiler] Section

This section documents the entries that can appear in an [XXX_Compiler] section of a *project.ini* file.

NOTE XXX is a placeholder for the name of the actual backend. For example, for the HC12 compiler, the name of this section would be [HC12_Compiler].

RecentCommandLineX

NOTE x is a placeholder for an integer.

Arguments

String with a command line history entry, e.g. “fibonacci.c”

Description

This list of entries contains the content of the command line history.

Saved

Only with Appearance set in the File->Configuration Save Configuration dialog.

CurrentCommandLine

Arguments

String with the command line, e.g. "fibonacci.c -w1"

Description

The currently visible command line content.

Saved

Only with Appearance set in the File->Configuration Save Configuration dialog.

StatusbarEnabled

Arguments

1/0

Special

This entry is only considered at startup. Later load operations do not use it any more.

Description

Is status bar currently enabled.

- 1: The status bar is visible
- 0: The status bar is hidden

Saved

Only with Appearance set in the File->Configuration Save Configuration dialog.

ToolbarEnabled

Arguments

1/0

Special

This entry is only considered at startup. Later load operations no longer use it.

Description

Is the toolbar currently enabled.

1: The toolbar is visible

0: The toolbar is hidden

Saved

Only with Appearance set in the File->Configuration Save Configuration dialog.

WindowPos

Arguments

10 integers, e.g. “0,1,-1,-1,-1,-1,390,107,1103,643”

Special

This entry is only considered at startup. Later load operations do not use it any more.

Changes of this entry do not show the “*” in the title.

Description

This number contains the position and the state of the window (maximized) and other flags.

Saved

Only with Appearance set in the File->Configuration Save Configuration dialog.

WindowFont

Arguments

size: == 0 -> generic size, < 0 -> font character height, > 0 font cell height
weight: 400 = normal, 700 = bold (valid values are 0 – 1000)
italic: 0 == no, 1 == yes
font name: max 32 characters.

Description

Font attributes.

Saved

Only with Appearance set in the File->Configuration Save Configuration dialog.

Example

```
WindowFont=-16,500,0,Courier
```

Options

Arguments

-W2

Description

The currently active option string. This entry is quite long as the messages are also stored here.

Saved

Only with Options set in the File->Configuration Save Configuration dialog.

EditorType

Arguments

0/1/2/3

Description

This entry specifies which editor configuration is active.

0: Global editor configuration (in the file mcutools.ini)

1: Local editor configuration (the one in this file)

2: Command line editor configuration, entry EditorCommandLine

3: DDE editor configuration, entries beginning with EditorDDE

For details see [Editor Configuration](#).

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

EditorCommandLine

Arguments

Command line for the editor.

Description

Command line content to open a file. For details see [Editor Configuration](#).

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

EditorDDEClientName

Arguments

Client command, e.g. “[open(%f)]”

Description

Name of the client for DDE editor configuration. For details see a [Editor Configuration](#).

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

EditorDDETTopicName

Arguments

Topic name. For example, “system”

Description

Name of the topic for DDE editor configuration. For details see [Editor Configuration](#).

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

EditorDDESvcName

Arguments

Service name. For example, “system”

Description

Name of the service for DDE editor configuration. For details see [Editor Configuration](#).

Saved

Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Example

[Listing C.1](#) shows a typical configuration file layout (usually *project.ini*):

Listing C.1 A Typical Local Configuration File Layout

```
[Editor]
Editor_Name=notepad
Editor_Exe=C:\windows\notepad.exe
Editor_Opts=%f

[XXX_Compiler]
StatusBarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
Options=-w1
EditorType=3
RecentCommandLine0=fibo.c -w2
RecentCommandLine1=fibo.c
CurrentCommandLine=fibo.c -w2
EditorDDEClientName=[open(%f)]
EditorDDETTopicName=system
EditorDDEServiceName=msdev
EditorCommandLine=C:\windows\notepad.exe %f
```

Local Configuration File Entries

Example

Index

Symbols

-! 123
390
390, 427
#asm 403
#define 192, 389
#elif 389
#else 389
#endasm 403
#endif 389
#error 389, 391, 748
#if 389
#ifdef 389
#ifndef 389
#include 194, 389
#line 389
#pragma 389
 CODE_SECTION 416
 CODE_SEG 341, 416, 519
 CONST_SECTION 136, 416
 CONST_SEG 344, 416
 CREATE_ASM_LISTING 347
 DATA_SECTION 416
 DATA_SEG 348, 416
 FAR 519
 INLINE 240, 351
 INTO_ROM 136, 352, 936
 LINK_INFO 354
 LOOP_UNROLL 165, 356
 mark 357
 MESSAGE 359
 NEAR 519
 NO_ENTRY 361, 514, 522, 523
 NO_EXIT 363, 514
 NO_FRAME 365, 514
 NO_INLINE 367
 NO_LOOP_UNROLL 165, 369
 NO_STRING_CONSTR 372, 427
 ONCE 373
 OPTION 328, 374
 pop 377
 push 379
 REALLOC_OBJ 381
 SHORT 519
STRING_SEG 383
TEST_CODE 385
TRAP_PROC 387, 402, 514, 515
 SAVE_ALL_REGS 515
 SAVE_NO_REGS 515
#pragma CONST_SEG 915
#pragma DATA_SEG 915
#pragma section 915
#undef 389
#warning 390, 391, 754
\$ 391
\$() 91
\${} 91
%(ENV) 121
%' 121
%' 121
%currentTargetName 44
%E 121
%e 121
%f 121
%N 121
%n 121
%p 121
%projectFileDir 44
%projectFileName 44
%projectFilePath 43
%projectSelectedFiles 44
%sourceFileDir 43
%sourceFileName 43
%sourceFilePath 43
%sourceLineNumber 43
%sourceSelection 43
%sourceSelUpdate 43
%symFileDir 44
%symFileName 44
%symFilePath 44
%targetFileDir 44
%targetFileName 44
%targetFilePath 44
*.bbl 45
*.pj 937
*.wpj 937
.abs 42

.c 113
.cpp 113
.cxx 113
.h 113
.ini 65
.lcf 918
.lib 42
.lst 797
.mcp 938
.o 114
/wait 60
@ “SegmentName” 393
@address 391
@bool 920
@far 920
@interrupt 922
@tiny 920
alignof 390, 401
ARCHIMEDES 325
_asm 126, 390, 403
_BIG_ENDIAN_ 326
_BITFIELD_TYPE_SIZE_REDUCTION_ 132
_BITFIELD_LSBIT_FIRST_ 128, 332, 335, 919
_BITFIELD_LSBYTE_FIRST_ 128, 335, 919
_BITFIELD_LSWORD_FIRST_ 128, 332, 335
_BITFIELD_MSBIT_FIRST_ 128, 332, 335, 919
_BITFIELD_MSBYTE_FIRST_ 128, 332, 335, 919,
920
_BITFIELD_MSWORD_FIRST_ 128, 332, 335
_BITFIELD_NO_TYPE_SIZE_REDUCTION_ 132,
333
_BITFIELD_TYPE_SIZE_REDUCTION_ 333
_CHAR_IS_16BIT_ 269, 335
_CHAR_IS_32BIT_ 269, 335
_CHAR_IS_64BIT_ 269, 336
_CHAR_IS_8BIT_ 269, 335
_CHAR_IS_SIGNED_ 269, 335
_CHAR_IS_UNSIGNED_ 269, 335
CNI 147, 327
_CODE_SEG 341, 344, 348, 383
_cplusplus 134, 327, 438, 440
DATE 325
_DEMO_MODE_ 326
_DIRECT_SEG 341, 344, 348, 383
_DOUBLE_IS_DSP_ 270, 337
_DOUBLE_IS_IEEE32_ 270, 337
_DOUBLE_IS_IEEE64_ 270, 337
_ELF_OBJECT_FILE_FORMAT_ 179, 331
_ENUM_IS_16BIT_ 269, 336
_ENUM_IS_32BIT_ 270, 336
_ENUM_IS_64BIT_ 270, 336
_ENUM_IS_8BIT_ 269, 336
_ENUM_IS_SIGNED_ 270, 336
_ENUM_IS_UNSIGNED_ 270, 336
_far 390, 395, 507
 Arrays 396
 Keyword 396
_FAR_SEG 341, 344, 348, 383, 716, 717
FILE 325
_FLOAT_IS_DSP_ 270, 336
_FLOAT_IS_IEEE32_ 270, 336
_FLOAT_IS_IEEE64_ 270, 336
HIWARE 325
_HIWARE_OBJECT_FILE_FORMAT_ 179, 331
_INT_IS_16BIT_ 269, 336
_INT_IS_32BIT_ 269, 336
_INT_IS_64BIT_ 269, 336
_INT_IS_8BIT_ 269, 336
_Interrupt 390
_interrupt 402
LINE 325
_LITTLE_ENDIAN_ 326
_LONG_DOUBLE_IS_DSP_ 270, 337
_LONG_DOUBLE_IS_IEEE32_ 270, 337
_LONG_DOUBLE_IS_IEEE64_ 270, 337
_LONG_IS_16BIT_ 270, 336
_LONG_IS_32BIT_ 270, 336
_LONG_IS_64BIT_ 270, 336
_LONG_IS_8BIT_ 270, 336
_LONG_LONG_DOUBLE_DSP_ 270, 337
_LONG_LONG_DOUBLE_IS_IEEE32_ 270, 337
_LONG_LONG_DOUBLE_IS_IEEE64_ 270, 337
_LONG_LONG_IS_16BIT_ 270, 336
_LONG_LONG_IS_32BIT_ 270, 336
_LONG_LONG_IS_64BIT_ 270, 336
_LONG_LONG_IS_8BIT_ 270, 336
MWERKS 325
_near 390, 400, 507
_NEAR_SEG 341, 344, 348, 383
_NO_RECURSION_ 337
_OPTIMIZE_FOR_SIZE_ 227, 327
_OPTIMIZE_FOR_TIME_ 227, 327

__OPTIMIZE_REG__ 338
__OPTION_ACTIVE__ 327
__PLAIN_BITFIELD_IS_SIGNED__ 270, 334, 335, 337
__PLAIN_BITFIELD_IS_UNSIGNED__ 270, 334, 335, 337
__PRODUCT_HICROSS__ 326
__PRODUCT_HICROSS_PLUS__ 326
__PRODUCT_SMILE_LINE__ 326
__PTR_SIZE_1__ 337
__PTR_SIZE_2__ 337
__PTR_SIZE_3__ 337
__PTR_SIZE_4__ 338
__PTRDIFF_T_IS_CHAR__ 330
__PTRDIFF_T_IS_INT__ 330, 331
__PTRDIFF_T_IS_LONG__ 330, 331
__PTRDIFF_T_IS_SHORT__ 330
__PTRMBR_OFFSET_IS_16BIT__ 270
__PTRMBR_OFFSET_IS_32BIT__ 270
__PTRMBR_OFFSET_IS_64BIT__ 270
__PTRMBR_OFFSET_IS_8BIT__ 270
__SHORT_IS_16BIT__ 269, 336
__SHORT_IS_32BIT__ 269, 336
__SHORT_IS_64BIT__ 269, 336
__SHORT_IS_8BIT__ 269, 336
__SHORT_SEG 344, 348, 417, 498
__SIZE_T_IS_UCHAR__ 329, 330
__SIZE_T_IS_UINT__ 329, 330
__SIZE_T_IS ULONG__ 329, 330
__SIZE_T_IS USHORT__ 329, 330
__STDC__ 126, 325, 327
__TIME__ 325
__TRIGRAPHS__ 141, 327
__va_sizeof__ 390, 401
__VERSION__ 326
__VTAB_DELTA_IS_16BIT__ 270, 337
__VTAB_DELTA_IS_32BIT__ 270, 337
__VTAB_DELTA_IS_64BIT__ 270, 337
__VTAB_DELTA_IS_8BIT__ 270, 337
__WCHAR_T_IS_UCHAR__ 329
__WCHAR_T_IS_UINT__ 329
__WCHAR_T_IS ULONG__ 330
__WCHAR_T_IS USHORT__ 329
__XGATE__ 337
_asm 390, 921
_bffo 516

_carry 517
_csem 516
_IOFBF 817
_IOLBF 817
_IONBF 817
_OVERLAP 717
_ovfl 517
_par 516
_PRESTART 717
_rol 517
_ror 517
_sif 518
_sifl 518
_sse 516
{Compiler} 91
{Project} 91
{System} 91

Numerics

0b 391

A

abort 798, 807, 822
About 45, 46, 47, 48, 50, 51
About Box 85
abs 804, 823
absolute assembly application 51
Absolute Functions 394
absolute variables 391
Absolute Variables and Linking 394
ABSPATH 75
abstract classes 478
acos 803, 823
acosf 804, 823
Activate Browser 43
-AddIncl 125
Alignment 509
alloc.c 798
-Ansi 126, 325, 327, 404
ANSI-C 147, 148, 328
 Reference Document 389
 Standard 389
ANSI-C 389
Application File Name 50
Argument 510

Array
 __far 396
Arrays with unknown size 921
asctime 808, 824
asin 803, 824
asinf 804, 824
asm 390, 403, 521
Assembler 521
assert 801, 825
assert.h 819
Associativity 945
atan 803, 825
atan2 803, 826
atan2f 804, 826
atanf 804, 825
atexit 798
atexit 807, 827
atof 809, 827
atoi 809, 828
atol 809, 828
auto 389
axgate.exe 37

B

batch file 60
-BfaB 128, 335
-BfaGapLimitBits 130
-BfaTSR 132
-BfaTSROFF 334
-BfaTSRON 334
Big Endian 326
bin 41
Binary Constants 391
binplugins 41
Bit Fields 414, 509, 919
Branch Optimization 420
Branch Sequence 422
Branch Tree 422
break 389
browse information 43
Browser 43
bsearch 805, 829
BUFSIZ 817
Burner 46
burner.exe 37

C
-C++ 134, 327
C++ 54, 387, 442
C++ comments 126, 150
C++ Constructor calls 797
C++ Destructor calls 797
C++ Front End 437
C++ Library 905
 I/O library 906
 Memory library 909
 String library 909
-C++c 134
-C++e 134
-C++f 134
C1 529
C1000 532
C10000 791
C1001 533
C10010 791
C10011 791
C10012 791
C10013 791
C1002 533
C10020 791
C10021 791
C10022 791
C10023 791
C10024 791
C10025 791
C10026 791
C10027 791
C10028 791
C10029 791
C1003 534
C10030 791
C10031 791
C10032 791
C10033 791
C10034 791
C10035 791
C1004 534
C1005 534
C1006 535
C1007 535
C1008 536

C1009	536	C1055	559
C1010	537	C1056	559
C1012	537	C1057	560
C1013	538	C1058	560
C1014	539	C1059	561
C1015	539	C1060	561
C1016	540	C1061	561
C1017	540	C1062	562
C1018	541	C1063	562
C1019	541	C1064	563
C1020	542	C1065	563
C1021	542	C1066	564
C1022	543	C1067	564
C1023	543	C1068	565
C1024	544	C1069	565
C1025	544	C1070	566
C1026	545	C1071	566
C1027	545	C1072	567
C1028	546	C1073	567
C1029	546	C1074	568
C1030	546	C1075	568
C1031	547	C1076	569
C1032	547	C1077	569
C1033	548	C1078	570
C1034	548	C1080	570
C1035	549	C1081	571
C1036	549	C1082	571
C1037	550	C1084	572
C1038	550	C1085	572
C1039	551	C1086	572
C1040	551	C1087	573
C1041	552	C1088	573
C1042	552	C1089	574
C1043	553	C1090	574
C1044	553	C1091	574
C1045	554	C1092	575
C1046	554	C1093	575
C1047	555	C1094	575
C1048	555	C1095	576
C1049	556	C1096	576
C1050	556	C1097	576
C1051	557	C1098	577
C1052	557	C1099	577
C1053	558	C1100	578
C1054	558	C1101	578

C1102	579	C1143	604
C1103	579	C1144	605
C1104	580	C12002	791
C1105	580	C12004	791
C11051	791	C12005	791
C1106	581	C12006	791
C11064	791	C12007	791
C11065	791	C12008	791
C11066	791	C12011	791
C1107	581	C12012	791
C1108	581	C12020	791
C1109	582	C12021	791
C1110	583	C12022	791
C1111	583	C12023	791
C1112	584	C12024	791
C1113	584	C12025	791
C1114	584	C12026	791
C1115	585	C12027	791
C1116	585	C12028	791
C1117	586	C12040	791
C1118	586	C12041	791
C1119	587	C12042	791
C1120	587	C12043	791
C1121	588	C12046	791
C1122	589	C12050	791
C1123	590	C12051	791
C1124	591	C12052	791
C1125	591	C12053	791
C1126	592	C12054	791
C1127	593	C12055	791
C1128	593	C12056	791
C1129	594	C12057	791
C1130	595	C12058	791
C1131	595	C12062	791
C1132	596	C12099	791
C1133	597	C12100	791
C1134	598	C12101	791
C1135	598	C12102	791
C1136	599	C12103	791
C1137	600	C13000	791
C1138	600	C13001	791
C1139	601	C13002	791
C1140	602	C13003	791
C1141	603	C13004	791
C1142	603	C13005	791

C13006	791	C13164	791
C13007	791	C13165	791
C13008	791	C13166	791
C13009	791	C13167	791
C13010	791	C13168	791
C13011	791	C13169	791
C13012	791	C13170	791
C13013	791	C13171	791
C13014	791	C13172	791
C13015	791	C13173	791
C13016	791	C13174	791
C13017	791	C13175	791
C13018	791	C13200	791
C13019	791	C13201	791
C13020	791	C13202	791
C13021	791	C13250	791
C13022	791	C13300	791
C13023	791	C13301	791
C13024	791	C13302	791
C13025	791	C13303	791
C13026	791	C13350	791
C13027	791	C13400	791
C13028	791	C13450	791
C13029	791	C13451	791
C13030	791	C13452	791
C13031	791	C13453	791
C13032	791	C13454	791
C13033	791	C13500	791
C13034	791	C1390	605
C13035	791	C1391	606
C13100	791	C1392	608
C13150	791	C1393	608
C13151	791	C1395	609
C13152	791	C1396	611
C13153	791	C1397	612
C13154	791	C1398	613
C13155	791	C1400	614
C13156	791	C1401	614
C13157	791	C1402	615
C13158	791	C1403	615
C13159	791	C1404	616
C13160	791	C1405	616
C13161	791	C1406	617
C13162	791	C1407	617
C13163	791	C1408	617

C1409	618	C17300	791
C1410	618	C17302	791
C1411	618	C17400	791
C1412	619	C1800	639
C1413	619	C18000	791
C1414	620	C1801	639
C1415	620	C1802	640
C1416	621	C1803	641
C1417	621	C1804	641
C1418	622	C1805	641
C1419	622	C1806	642
C1420	623	C1807	642
C14201	791	C1808	643
C14208	791	C1809	644
C14209	791	C1810	644
C1421	623	C18100	791
C14210	791	C18103	791
C1422	624	C1811	645
C1423	625	C18113	791
C1424	625	C1812	645
C1425	626	C1813	645
C1426	626	C1814	646
C1427	626	C1815	646
C1428	627	C1816	647
C1429	627	C1817	648
C1430	628	C1819	648
C1431	628	C1820	649
C1432	629	C1821	649
C1433	629	C1822	650
C1434	630	C1823	650
C1435	630	C1824	651
C1436	631	C1825	652
C1437	632	C1826	652
C1438	633	C1827	653
C1439	634	C1828	653
C1440	635	C1829	654
C1441	635	C1830	654
C1442	636	C1831	655
C1443	636	C1832	655
C1444	637	C1833	656
C1445	638	C1834	656
C15251	791	C1835	657
C15402	791	C1836	657
C17000	791	C1837	658
C17001	791	C1838	658

C1839	658	C2018	679
C1840	659	C2019	680
C1842	659	C2020	680
C1843	660	C2021	681
C1844	660	C2022	682
C1845	660	C2023	683
C1846	661	C2024	683
C1847	661	C2025	684
C1848	662	C21000	791
C1849	662	C21001	791
C1850	663	C21002	791
C1851	663	C21003	791
C1852	664	C21004	791
C1853	664	C21005	791
C1854	665	C21006	791
C1855	665	C21007	791
C1856	666	C21008	791
C1857	667	C21009	791
C1858	667	C21010	791
C1859	668	C21011	791
C1860	669	C21012	791
C1861	669	C21013	791
C18700	791	C21015	791
C2	529	C21016	791
C2000	670	C2200	684
C20000	791	C22000	792
C20001	791	C22001	792
C2001	670	C22002	793
C2004	671	C2201	685
C2005	671	C2202	685
C2006	672	C2203	686
C20062	791	C2204	687
C2007	672	C2205	688
C2008	673	C2206	688
C2009	673	C2207	689
C2010	674	C2209	690
C20100	791	C2210	690
C2011	675	C2211	691
C20110	791	C2401	691
C2012	676	C2402	692
C2013	676	C2450	692
C2014	677	C2550	693
C2015	678	C2700	694
C2016	678	C2701	694
C2017	679	C2702	695

C2703	695	C4003	719
C2704	696	C4004	720
C2705	696	C4005	721
C2706	697	C4006	721
C2707	698	C4100	722
C2708	698	C4101	723
C2709	699	C4200	723
C2800	699	C4201	724
C2801	700	C4202	724
C2802	700	C4203	725
C2803	701	C4204	726
C2900	701	C4205	726
C2901	702	C4300	727
C3000	702	C4301	727
C3100	703	C4302	728
C3200	704	C4303	729
C3201	704	C4400	729
C3202	705	C4401	730
C3300	705	C4402	730
C3301	706	C4403	731
C3302	706	C4404	731
C3303	707	C4405	732
C3304	708	C4406	732
C3400	708	C4407	732
C3401	709	C4408	733
C3500	709	C4409	733
C3501	710	C4410	734
C3600	710	C4411	734
C3601	711	C4412	735
C3602	711	C4413	736
C3603	712	C4414	736
C3604	712	C4415	737
C3605	713	C4416	737
C3606	713	C4417	737
C3700	714	C4418	738
C3701	714	C4419	739
C3800	715	C4420	739
C3801	715	C4421	740
C3802	716	C4422	740
C3803	716	C4423	741
C3804	717	C4424	741
C3900	717	C4425	742
C4000	718	C4426	742
C4001	718	C4427	743
C4002	719	C4428	743

C4429	744	C5352	764
C4430	744	C5353	765
C4431	745	C5354	765
C4432	745	C5355	766
C4433	746	C5356	766
C4434	746	C5400	767
C4435	746	C5401	767
C4436	747	C5403	767
C4437	747	C5500	767
C4438	748	C5650	767
C4439	748	C5651	767
C4440	749	C5660	768
C4441	749	C5661	768
C4442	750	C5662	769
C4443	750	C5680	769
C4444	751	C5681	769
C4445	752	C5682	769
C4446	752	C5683	770
C4447	753	C5684	770
C4448	754	C5685	770
C4449	754	C5686	771
C4700	755	C5687	771
C4701	755	C5688	772
C4800	755	C5689	772
C4801	756	C5690	772
C4802	756	C5691	773
C4900	756	C5692	773
C50	529	C5693	773
C5000	757	C5700	774
C5001	757	C5701	774
C5002	758	C5702	775
C5003	758	C5703	776
C5004	759	C5800	776
C5005	759	C5900	777
C5006	759	C5901	777
C51	530	C5902	778
C5100	760	C5903	778
C52	530	C5904	779
C5200	761	C5905	779
C5250	761	C5906	780
C5300	762	C5907	780
C5302	762	C5908	781
C5320	763	C5909	781
C5350	763	C5910	782
C5351	764	C5911	783

C5912 783
C5913 784
C5914 785
C5915 785
C5916 786
C5917 787
C5918 787
C5919 788
C5920 789
C5921 789
C6000 790
C6001 790
C6002 791
C64 530
C65 531
C66 532
Call Protocol 510
Caller/Callee Saved Registers 523
calloc 798, 805, 830
carry flag 517
case 389
catch 439
-Cc 118, 136, 343, 345, 346, 349, 350, 351, 352, 353,
 368, 372, 418, 936, 938
-Cctv 156
-Ccx 138, 913, 917
ceil 803, 831
ceilf 804, 831
char 389, 404
CHAR_BIT 813
CHAR_MAX 813
CHAR_MIN 813
-Ci 141, 327
class 439, 451
clearerr 810, 831
ClientCommand 70
clock 808, 832
clock_t 818
CLOCKS_PER_SEC 819
-Cn 145
-CnCtr 554
-Cni 147, 327
CODE 119, 341, 344, 348, 383
CODE GENERATION 120
Code Size 407
CODE_SECTION 341, 416
CODE_SEG 341, 416
CodeWarrior 37, 41, 72, 932, 936, 938
CodeWright 69
Codewright 937
color 284, 285, 286, 287, 288
COM 41, 72
Command Line Arguments 44, 46, 47, 48, 49, 50
comments 921
Common Source Files 796
compactC++ 54
compactC++ library 905
Compiler
 Configuration 65
 Control 81
 Error
 Messages 84
 Error Feedback 86
 Graphic Interface 59
 Include file 113
 Input File 85, 113
 Menu 76
 Menu Bar 64
 Messages 82
 Option 79
 Option Settings Dialog 79
 Standard Types Dialog Box 78
 Status Bar 64
 Tool Bar 63
 User Interface 59
compiler generated functions 441, 459
Compiler Pragmas 339
COMPOPTIONS 92, 96, 117
const 136, 389, 423
CONST_SECTION 136, 344, 416
CONST_SEG 344, 416
Constant Function 500
constant objects 442
constructor 456
continue 389
conversion
 by constructor 458
 functions 458
COPY 717
Copy Down 797
copy down 395
Copy Template 50
Copying Code from ROM to RAM 932

COPYRIGHT 97	Directive #define 192, 389 #elif 389 #else 389 #endif 389 #error 389 #if 389 #ifdef 389 #ifndef 389 #include 194, 389 #line 389 #pragma 389 #undef 389 Preprocessor 389
cos 803, 832	Display generated command lines in message window 45, 46, 47, 48, 49, 51
cosf 804, 832	div 804, 834
cosh 803, 833	div_t 818
coshf 804, 833	Division 331, 404
Cosmic 913	do 389
-Cppc 150	DOS 123
-Cq 152	double 389
CREATE_ASM_LISTING 347	download 395
-CsIni0 154	
-CswMaxLF 157	
-CswMinLB 159	
-CswMinLF 161	
-CswMinSLB 163, 929	
ctime 808, 833	
CTRL-S 76	
ctype 800	
ctype.h 820	
-Cu 118, 165, 356, 369	
Current Directory 90, 98	
CurrentCommandLine 960	
-Cx 168	
cxgate.exe 37	
D	E
-D 169, 192	EABI 334
DATA_SECTION 348, 416	EBNF 941
DATA_SEG 348, 416, 498	-Ec 171
decoder.exe 37	EC++ 54
default 389	Editor 957
default arguments 485	Editor_Exe 954, 958
Default Directory 949	Editor_Name 953, 957
DEFAULT.ENV 89, 90, 98, 99, 109	Editor_Opts 954, 958
default.env 117	EditorCommandLine 963
DEFAULT_RAM 717	EditorDDEClientName 964
DEFAULT_ROM 717	EditorDDEServiceName 964
DEFALUTDIR 90, 98, 113, 949	EditorDDETTopicName 964
DefaultDir 949	EditorType 963
define 169	EDOM 811
defined 390	EDOUT 114
delete 439, 485	-Eencrypt 173
destructor 457	EEPROM 925
difftime 808, 834	-Ekey 175
DIG 812	ELF/DWARF 56, 78, 394, 936
DIRECT 341, 344, 348, 383	else 389
	Embedded Application Binary Interface 334

Endian 326
ENTRIES 394
enum 389
-Env 90, 177
ENVIRONMENT 89, 99
Environment
 COMPOPTIONS 96, 117
 COPYRIGHT 97
 DEFAULTDIR 90, 98, 113, 949
 ENVIRONMENT 90, 99
 ENVIRONMENT 89
 ERRORFILE 100, 114
 File 89
 GENPATH 102, 105, 107, 113, 184
 HICOMPOPTIONS 96
 HIENVIRONMENT 99
 HIPATH 102, 107
 INCLUDETIME 97, 103
 LIBPATH 102, 105, 110, 113, 114, 184, 185
 LIBRARYPATH 105, 113, 114, 184, 185
 OBJPATH 107, 114, 230
 TEXTPATH 108, 186, 188, 192, 194, 203, 205, 210
 TMP 109
 USELIBPATH 110
 USERNAME 97, 111
 Variable 89, 95
Environment Variable 89, 339
 89
Environment Variables 75
EOF 817
EPROM 395
EPSILON 812
equates 914
ERANGE 811
errno 811
errno.h 811
Error
 Handling 801
 Listing 114
 Messages 84
Error Format
 Microsoft 292
 Verbose 292
Error Listing 114
ERRORFILE 100, 114
Escape Sequences 947
exit 798, 807, 835

EXIT_FAILURE 818
EXIT_SUCCESS 818
exp 803, 835
expf 804, 835
Explorer 59, 90
Extended Backus-Naur Form, see EBNF
extern 389, 442
extern "C" 439
external linkage 442

F

-F 936
-F1 56, 179, 331, 417
-F1o 179
-F2 56, 179, 331, 417
F2 64
-F2o 179
-F6 179
-F7 179
fabs 242, 803, 836
fabsf 242, 804, 836
FAR 341, 344, 348, 383, 519
far 390, 395
fclose 810, 836
feof 810, 837
ferror 810, 837
fflush 810, 838
fgetc 810, 839
fgetpos 811, 839
fgets 810, 840
-Fh 56, 179, 331
FILE 817
File
 Environment 89
 Include 113
 Object 114
 Source 113
File Manager 90
File Names 407
FILENAME_MAX 817
float 389
float.h 812
Floating Point 508
floor 803, 841
floorf 804, 841
FLT_RADIX 812

FLT_ROUNDS 812
fmod 803, 841
fmodf 804
fopen 810, 842
FOPEN_MAX 817
for 389
fpos_t 817
fprintf 811, 843
fputc 810, 844
fputs 810, 844
Frame, see Stack Frame
fread 810, 845
free 798, 805, 845
freopen 810, 846
frexp 803, 846
frexpf 804, 846
friend 439, 456
Front End 389
fscanf 811, 847
fseek 811, 847
fsetpos 811, 848
ftell 811, 849
FUNCS 717
function overloading 484
Function Pointer 509
fwrite 810, 849

G

GEN_FUNCS_SEGMENT 441
GENPATH 75, 102, 105, 107, 113, 184, 936
getc 810, 850
getchar 810, 850
getenv 807, 850
gets 810, 851
gmtime 808, 851
goto 389, 407
GUI Graphical User Interface 59

H

-H 182, 937, 939
HALT 797, 798
heap.c 798
Help 45, 46, 47, 48, 50, 51
Hexadecimal Constants 391
HICOMPOPTIONS 96

HIENVIRONMENT 99
HIPATH 102
hiwave.exe 37, 43
HOST 119, 120
How to Generate Library 795
HUGE_VAL 815

I

-I 113, 184, 936
I/O library 906
I/O Registers 395
Icon 59
ide.exe 37
IEEE 508
if 389
Implementation Restriction 404
Include Files 113, 407
INCLUDETIME 97, 103
inheritance 467
INLINE 240, 351
inline 240, 439, 444, 497
Inline Assembler, see Assembler
INPUT 119, 120
int 389
INT_MAX 813
INT_MIN 813
Intel 326
Internal ID's 407
Interrupt 402, 403, 922
 keyword 402
 vector 402
interrupt 388, 390, 507
Interrupt Procedure 514
Interrupt signaling 518
INTO_ROM 136, 352, 936
Intrinsics 515
IPATH 107
isalnum 806, 852
isalpha 806, 852
iscntrl 806, 852
isdigit 806, 852
isgraph 806, 852
islower 806, 852
isprint 806, 852
ispunct 806, 852, 853

isspace 806, 852, 853
isupper 806, 852, 853
isxdigit 806, 852, 853

J

jmp_buf 815
Jump Table 422

L

-La 186
Labels 407
labs 804, 853
LANGUAGE 120, 145
-Lasm 188
-Lasmc 190
Lazy Instruction Selection 519

lconv 813
ldexp 803, 854
ldexpf 804, 854
-Ldf 192, 325
ldiv 804, 854
ldiv_t 818
Lexical Tokens 407
-Li 194
libmaker 42
libmaker.exe 37
LIBPATH 75, 102, 105, 110, 113, 114, 184, 185
library 42
Library File Name 51
Library Files 795, 797
Library Structure 801
LIBRARYPATH 105, 113, 114, 184, 185
-Lic 196, 199
-LicA 197, 201
Limits
 Translation 404
limits.h 813
Line Continuation 94
LINK_INFO 354
linker.exe 37
Little Endian 326
-Ll 203, 929
-Lm 205
-LmCfg 207
-Lo 210

locale.h 813
localeconv 808, 855
Locales 800
localtime 808, 855
log 803, 856
log10 803, 856
log10f 804, 856
logf 804, 856
long 389
LONG_MAX 813
LONG_MIN 813
longjmp 807, 857
LOOP_UNROLL 165, 356
-Lp 212
-LpCfg 214
-LpX 216

M

Macro 169
 Predefined 325
Macro Expansion 407
maker.exe 37
malloc 798, 805, 857
MANT_DIG 812
mark 357
math.h 815, 879
MAX 812
MAX_10_EXP 812
MAX_EXP 812
MB_LEN_MAX 813, 818
mblen 798, 806, 858
mbstowcs 798, 806, 858
mbtowc 798, 806, 859
MCUTOOLS.INI 67, 91, 98
Member Access Control 453
member functions 452
memchr 802, 859
memcmp 802, 860
memcpy 242, 802, 861
memmove 802, 861
Memory library 909
memset 242, 802, 861
MESSAGE 120, 359
MESSAGES 119
Messages 44, 46, 47, 48, 49, 50

Messages Settings 82
Microsoft 292
Microsoft Developer Studio 70
Microsoft Visual Studio 51
MIN 812
MIN_10_EXP 812
MIN_EXP 812
Missing Prototype 921
mtime 808, 862
modf 803, 862
modff 804, 862
Modulus 331, 404
msdev 70
MS-DOS 123
multiple inheritance 464, 467

N

-N 217
name encoding 492
NAMES 936
NEAR 341, 344, 348, 383, 519
near 390, 400
new 439, 485
New Project Wizard 913
NO_ENTRY 361, 514, 522
NO_EXIT 363, 514
NO_FRAME 365, 514
NO_INIT 926
NO_INLINE 367
NO_LOOP_UNROLL 165, 369
NO_STRING_CONSTR 372, 427
-NoBeep 219
-NoDebugInfo 220
-NoEnv 222
-NoPath 224
NULL 817
Numbers 407

O

-Oa 225
Object
 File 114
-ObjN 229
OBJPATH 75, 107, 114, 230
-Oc 231

-Od 233
-Odb 235
-OdocF 118, 120, 237, 929
-Odocf 328
offsetof 817
-Oi 118, 240, 497
-Oilib 242
-OnBRA 245
ONCE 373
-OnCopyDown 248
-OnCstVar 250
-OnPMNC 252
-Ont 254, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 789
operator 439
 # 390
 ## 390
 defined 390
operator delete 485
operator new 485
operator overloading 484
OPTIMIZATION 119, 120
Optimization
 Branches 420
 Lazy Instruction Selection 519
 Shift optimizations 420
 Strength Reduction 420
 Time vs. Size 227
 Tree Rewriting 421
OPTION 374
Option
 CODE 119
 CODE GENERATION 120
 HOST 119, 120
 INPUT 119, 120
 LANGUAGE 120, 145
 LANGUAGE 119
 MESSAGE 120
 MESSAGES 119
 OPTIMIZATION 119, 120
 OUTPUT 119, 120
 STARTUP 119
 TARGET 119
 VARIOUS 119, 120
Options 45, 46, 47, 48, 49, 50, 949, 962
-Or 118, 497
-Os 227, 327, 422

-Ot 227, 327
OUTPUT 119, 120
overflow flag 517
overloading 484

P

Parameter 510
 Register 510
Parsing Recursion 407
Path List 93
-Pe 261
perror 801, 863
-Pio 263
piper.exe 37
PLACEMENT 915
Pointer
 far 396
 Compatibility 400
Pointer Type 509
pop 377
pow 803, 863
powf 804, 863
Precedence 945
Predefined Macro 325
Premia 69
Preprocessor
 Directives 389
printf 809, 811, 864
printf.c 799
private 439, 453
Procedure
 Call Protocol 510
 Interrupt 514
 Return Value 511
 Stack Frame 511
 Variable, see Function Pointer
-Prod 38, 93, 265
project.ini 93, 96, 117
protected 439, 454
ptrdiff_t 328, 816
public 439, 454
pure virtual functions 478
push 379
putc 810, 864
putchar 810, 865
puts 810, 865

PVCS 110

Q

qsort 805, 866
-Qvpt 267

R

raise 807, 867
RAM 932
rand 805, 867
RAND_MAX 818
realloc 798, 805, 868
REALLOC_OBJ 381
RecentCommandLine 959
Recursive comments 921
Register
 Parameter 510
register 389
Register initialisation 797
regservers.bat 41
remove 810, 869
rename 810, 869
Restriction
 Implementation 404
return 389
Return Value 511
rewind 811, 870
RGB 284, 285, 286, 287, 288
ROM 423, 932, 936
ROM libraries 797
ROM_VAR 118, 136, 137, 418, 717

S

SAVE_ALL_REGS 515
SAVE_NO_REGS 515
SaveAppearance 950
SaveEditor 951
SaveOnExit 950
SaveOptions 951
scanf 811, 870
SCHAR_MAX 813
SCHAR_MIN 813
SEEK_CUR 817
SEEK_END 817
SEEK_SET 817

Segment 518
Segmentation 415
Semaphore 515
Service Name 70
setbuf 810, 871
setjmp 807, 871
setjmp.h 815
setlocale 808, 872
setvbuf 810, 873
Shift optimizations 420
SHORT 344, 348, 519
short 389
-ShowAboutDialog 38
-ShowBurnerDialog 38
ShowConfigurationDialog 38
-ShowMessageDialog 38
-ShowOptionDialog 38
-ShowSmartSliderDialog 38
ShowTipOfDay 952
SHRT_MAX 813
SHRT_MIN 813
sig_atomic_t 816
SIG_DFL 816
SIG_ERR 816
SIG_IGN 816
SIGABRT 816
SIGFPE 816
SIGILL 816
SIGINT 816
signal 807, 873
signal.c 798
signal.h 816
Signals 798
signed 389
SIGSEGV 816
SIGTERM 816
sin 803, 874
sinf 804, 874
single inheritance 463
single object file 51
sinh 803, 875
sinhf 804
Size
 Type 507
size_t 328, 816
sizeof 389
Smart Control 81
Smart Sliders 47
Source File 113
special member functions 456
Special Modifiers 121
sprintf 809, 875
sqrt 803, 879
sqrtf 804, 879
srand 805, 879
sscanf 809, 880
SSTACK 717
Stack
 Frame 511
Standard Types 78
start 60
STARTUP 119, 717
startup 93
Startup Files 796
startup option 38
startup.c 797
static 389
static linkage 442
static members 452
StatusbarEnabled 960
stdarg 401
stdarg.h 401, 819
stddef.h 816
stderr 818
stdin 818
stdio.h 817
stdlib. 798
stdlib.c 798
stdlib.h 818, 879, 903
stdout 320, 818
strcat 802, 883
strchr 802, 884
strcmp 802, 884
strcoll 808, 885
strcpy 242, 802, 885
strcspn 802, 886
Strength Reduction 420
strerror 801, 886
strftime 808, 887
String library 909
string.h 819
STRING_SECTION 383

STRING SEG	383	TipTimeStamp	953
STRINGS	717	TMP	109
Strings	395	TMP_MAX	817
strlen	242, 802, 888	TMP010_Chapter.fm, using this file to create the first chapter in a book	37
strncat	802, 889	tmpfile	810, 898
strcmp	802, 889	tmpnam	810, 899
strcpy	802, 890	tolower	806, 899
strpbrk	802, 890	ToolbarEnabled	961
strrchr	802, 891	Topic Name	70
strspn	802, 891	toupper	806, 900
strstr	802, 892	Translation Limits	404
strtod	809, 892	TRAP_PROC	387, 402, 514, 515, 922
strtok	802, 893	try	439
strtol	809, 894	Type	
strtoul	809, 895	Alignment	509
struct	389	Bit Fields	509
strxfrm	808, 896	Floating Point	508
support@metrowerks.com	941	Pointer	509
support_europe@metrowerks.com	941	Size	507
switch	389	Type Declarations	407
synchronization	60	Type Sizes	47
system	807, 896	typedef	389

T

-T	78, 269, 507, 508, 727, 728, 757, 758, 929
tan	803, 897
tanf	804, 897
tanh	803, 897
tanhf	804, 897
TARGET	119
Target Settings Preference Panel	42
template	439
templates	475
termination	60
TEST_CODE	385
TEST_ERROR	755
TEXTPATH	75, 108, 186, 188, 192, 194, 203, 205, 210, 212
this	439
throw	439
time	808, 898
time.h	818
time_t	818
Tip of the Day	60
TipFilePos	952

U

UCHAR_MAX	813
UINT_MAX	813
ULONG_MAX	813
UltraEdit	70
ungetc	810, 900
union	389
UNIX	90
unsigned	389
Use custom PRM file	49
Use Decoder to generate Disassembly Listing	45, 47
Use template PRM file	49
Use third party debugger	43
USELIBPATH	110
USERNAME	97, 111
USHRT_MAX	813

V

-V	276
va_arg	401, 807, 901
va_end	807, 901

va_start 807, 901
Variable
 Environment 89
VARIOUS 119, 120
VECTOR 402
vfprintf 811, 901
-View 278
virtual 439, 448
virtual base classes 467
virtual function tables 441
virtual functions 459
VIRTUAL_TABLE_SEGMENT 440
Visual C++ 51
void 389
volatile 389, 414
vprintf 811, 901
vsprintf 809, 901

W

-W1 322
-W2 323, 918
wchar_t 328, 816
wctomb 798, 806, 903
wctomb 798, 806, 902
-WErrFile 280
while 389
WindowFont 962
WindowPos 961
Windows 90
WinEdit 101, 937
Winedit 69
-Wmsg8x3 282
-WmsgCE 284
-WmsgCF 285
-WmsgCI 286
-WmsgCU 287
-WmsgCW 288
-WmsgFb 87, 283, 289, 293, 295, 297, 299, 301
-WmsgFbi 289
-WmsgFbm 289
-WmsgFi 87, 283, 291, 292, 295, 297, 299, 301
-WmsgFim 292
-WmsgFiv 292
-WmsgFob 294, 297
-WmsgFoi 295, 296, 299, 301

-WmsgFonf 298
-WmsgFonp 295, 297, 299, 300, 301
-WmsgNe 302
-WmsgNi 304
-WmsgNu 305
-WmsgNw 307
-WmsgSd 308
-WmsgSe 310
-WmsgSi 312
-WmsgSw 314
-WOutFile 316
-Wpd 118, 318
-WStdout 320
www.metrowerks.com 941

Z

Zero Out 796
zero out 395

