HI-CROSS+

Motorola HC08 Compiler

May 24, 1998

The information in this document is based on
HC08 Compiler V5.0.06
or higher

Table Of Contents

Copyright and Trademarks 27
Copyright 27
Disclaimer 27
Trademarks. 27

Introduction. 29
Application Programs. 29
Literature 29
Highlights. 30
Structure of this Document 30
Specific Configuration Markers. 31
HI-CROSS+ vs. Smile~Line 31
C++, EC++, compactC++ 32
Object File Formats 33
HIWARE Object File Format 33
ELF/DWARF Object File Format. 33
Tools 34
Compiler 34
Decoder. 34
Linker 34
Simulator/Debugger 34
Mixing Object File Formats 34

HI-CROSS+ Graphical User Interface 35
Main Window. 36
Window Title 36
Content Area 36
Tool Bar 38
Status Bar 39
Menu Bar 39
File Menu 39
Edit Settings Dialog 40
Save Configuration Dialog 44
Compiler Menu 45
View Menu 46
Help Menu. 47
Standard Types Dialog Box 47

Advanced Options Dialog Box 48
Compiler Smart Control Dialog Box 50
Message Settings Dialog Box 51
Changing the Class associated with a Message 52
Retrieving Information about an Error Message. 53
About Box 54
Specifying the Input File 54
Use the Command Line in the Tool Bar to Compile 55
Compiling a New File 55
Compiling a File which has already been compiled 55
Use the Entry File - Compile... 55
Use Drag and Drop 55
Message/Error Feedback 55
Example: 56
Use Information from the Compiler Window 56
Use a User Defined Editor 56

Environment 57
The Current Directory 57
Global Initialization File (MCUTOOLS.INI) (PC only) 58
[Installation] Section 59
[Options] Section 59
[XXX_Compiler] Section 59
[Editor] Section 61
Example 61
Local Configuration File (usually project.ini) 62
[Editor] Section 63
[XXX_Compiler] Section 64
Example 68
Paths 68
Line Continuation 69
Environment Variable Details. 70
COMPOPTIONS: Default Compiler Options. 71
COPYRIGHT: Copyright Entry in Object File. 72
DEFAULTDIR: Default Current Directory 73
ENVIRONMENT: Environment File Specification 74
ERRORFILE: Error File Name Specification 75
GENPATH: #include “File” Path 77
INCLUDETIME: Creation Time in Object File 78
LIBRARYPATH: ‘include <File>’ Path 79
OBJPATH: Object File Path 80
TEXTFAMILY: Text Font Family 81

TEXTKIND: Text Font Character Set 82
TEXTPATH: Text File Path 83
TEXTSIZE: Text Font Size 84
TEXTSTYLE: Text Font Style 85
TMP: Temporary directory 86
USELIBPATH: Using LIBPATH Environment Variable 87
USERNAME: User Name in Object File. 88

Files. 89
Input Files. 89
Source Files 89
Include File 89
Output Files 90
Object Files 90
Error Listing 90
Interactive Mode (Compiler window open) 90
Batch Mode (Compiler window not open) 91

Compiler Options 93
Option Recommendation 94
Compiler Option Details 95
Option Groups 95
Option Scopes 96
Option Detail Description 96
-!: File Names to DOS Length 98
-Ansi: Strict ANSI 99
-BfaB: Bit Field Byte Allocation 100
-C++: Enable C++ Support 101
-Ca: 7bit Pointer Difference 102
-Cc: Allocate Constant Objects into ROM. 103
-Ci: Tri- and Bigraph Support 105
-Cn: Disable compactC++ features 108
-Cni: No Integral Promotion 110
-Cq: Propagate const and volatile Qualifiers for Structs 113
-CswMaxLB: Maximum Load Factor for Switch Tables 114
-CswMinLB: Minimum Number of Labels for Switch Tables 116
-CswMinSLB: Minimum Number of Labels for Search Switch Tables 118
-CswMinLB: Minimum Load Factor for Switch Tables 120
-Cu: Loop Unrolling. 122
-Cv: Local Variables of Size 2 in _OVERLAP 125

-Cx: No Code Generation 126
-D: Macro Definition 127
-Ec: Conversion from 'const T*' to 'T*' 128
-Env: Set Environment Variable 130
-F: Object File Format. 131
-Fd: Doubles are IEEE32 133
-H: Short Help. 134
-I: Include File Path. 135
-La: Generate Assembler Include File 136
-Li: List of Included Files 138
-Lic: License Information 139
-Lic: License Information about every Feature in Directory 140
-Ldf: Log Predefined Defines to File 141
-Ll: Statistics about each Function 143
-Lm: Include Files in Make File List 144
-Lo: Object File List 146
-Lp: Preprocessor Output 147
-M: Memory Model 148
-N: Display Notify Box. 149
-NoBeep: No Beep in Case of an Error. 150
-ObjN: Object file name specification 151
-Oc: Common Subexpression Elimination (CSE) 152
-OdocF: Dynamic Option Configuration for Functions. 153
-Oi: Inline Expansion of Functions. 155
-OnP: Disable peephole Optimization 157
-Ont: Disable Tree Optimizer 159
-Or: Lazy Register Optimization. 161
-Os/-Ot: Main Optimization Target 162
-Pe: Preprocessing Escape Sequences in Strings 163
-Pio: Include files only once 165
-Prod: specify project file at startup 167
-Qvtp: Qualifier for virtual table pointers. 168
-T: Flexible Type Management. 169
-V: Prints the Compiler Version 174
-Wmsg8x3: Cut file names in Microsoft format to 8.3. 175
-WmsgFb: Set message file format for batch mode 176
-WmsgFi: Set message format for interactive mode. 178
-WmsgFob: Message format for Batch Mode. 180
-WmsgFoi: Message Format for Interactive Mode. 182

-WmsgFonf: Message Format for no File Information 184
-WmsgFonp: Message Format for no Position Information 185
-WmsgNe: Number of Error Messages 187
-WmsgNi: Number of Information Messages 188
-WmsgNw: Number of Warning Messages 189
-WmsgSd: Setting a Message to Disable 190
-WmsgSe: Setting a Message to Error 191
-WmsgSi: Setting a Message to Information 192
-WmsgSw: Setting a Message to Warning. 193
-Wpd: Error for Implicit Parameter Declaration 194
-W1: No Information Messages 196
-W2: No Information and Warning Messages 197

Compiler Predefined Macros 199
Compiler Vendor Defines 199
Product Defines 199
Data Allocation Defines 200
Various Defines for Compiler Option Settings 201
ANSI-C Standard Types 'size_t' and 'ptrdiff_t' Defines. 201
Macros for HC08. 203
Object File Format Defines 203
Bit Field Allocation Defines 204
Macros for HC08. 205
Type Information Defines 205
Motorola HC08 Specific Defines. 207

Compiler Pragmas. 209
Pragma Details 209
#pragma CODE_SEG: Code Segment Definition 211
#pragma CONST_SEG: Constant Data Segment Definition. 214
#pragma CREATE_ASM_LISTING: Create an Assembler Include File Listing . 217
#pragma DATA_SEG: Data Segment Definition 218
#pragma INLINE: Inline next Function Definition 221
#pragma INTO_ROM: Put next Variable Definition into ROM 222
#pragma LOOP_UNROLL: Force Loop Unrolling. 223
#pragma MESSAGE: Message Setting 224
#pragma NO_ENTRY: No Entry Code 225
#pragma NO_EXIT: No Exit Code 226

#pragma NO_FRAME: No Frame Code 227
#pragma NO_LOOP_UNROLL: Disable Loop Unrolling 228
#pragma NO_RETURN: No Return Instruction 229
#pragma NO_STRING_CONSTR: No String Concatenation during Preprocessing 231
#pragma ONCE: Include Once 232
#pragma OPTION: Additional Options 233
#pragma STRING_SEG: String Segment Definition 235
#pragma TRAP_PROC: Mark Function as Interrupt Function 237

ANSI-C Front End 239
Implementation Features 239
Keywords 239
Preprocessor Directives 239
Language Extensions 239
Global variable address modifier (@address) 240
Examples 241
far Keyword 241
Usage of far for Pointers 242
far and arrays 242
far and typedef names 243
far and global variables 243
far and C++ classes 244
far and C++ references 244
Usage of far for Functions 245
near Keyword 246
rom Keyword 246
uni Keyword 247
Compatibility 247
interrupt Keyword 248
Implementation Defined Behavior 249
Right Shifts 249
Initialization of Aggregates with Non-Constants 249
Sign of char 249
Division and Modulus 249
Translation Limitations 250
ANSI-C Standard 251
Integral promotions 251
Signed/unsigned integers 251
Arithmetic conversions 251
Order of operand evaluation 252
Rules for standard type sizes 252
Floating Type Formats 253
Floating point representation of 500.0 for IEEE 253
Representation of 500.0 in IEEE32 format 254
Representation of 500.0 in IEEE64 format 255
Representation of 500.0 in DSP format 255
Volatile Objects, Absolute Variables 257

Bit Fields 257
Signed Bit Fields 257
Recommendations 258
Segmentation 258
Example for Segmentation Without Compiler Option -Cc 260
Example for Segmentation With Compiler Option -Cc 260
Optimizations 261
Peephole Optimizer 261
Strength Reduction 262
Shift Optimizations 262
Branch Optimizations 262
Dead Code Elimination 262
Constant Variable Optimization 262
Tree Rewriting 263
Switch Statements 263
Absolute Values 264
Combined Assignments 264
Usage of Qualifiers for Pointers 265
Defining C Macros Containing HLI Assembler Code 267
Defining a macro 267
Using Macro Parameters 268
Using Immediate Addressing Mode in HLI assembler Macros 269
Generating unique Labels in HLI Assembler Macros 269
Generating Assembler Include Files 270
General 271
. . . . . Macros :272
Enums 274
Types 274
Functions 275
Variables : 276
Comments 276

C++ Front End 279
Overview 279
Implementation 279
Features 279
Additional Keywords in C++ 281
C Linkage 281
The Standard Predefined Macro __cplusplus 282
Differences between ANSI-C and C++ 283
Empty Parameter Declarations 283
Return Values 283
Implicit Parameter Declaration 283
Constant Variables 283
Name Spaces 284
Empty Classes 284

More Details about C++ 284
C++ and Embedded Systems 284
About Compiler Generated Functions 285
Use Inlining. 285
Use Class Templates 285
Avoid Function Templates 286
Reduce the type for virtual function table delta 286
Avoid Pointer to Member Parameters and Pointer to member Returns 287
Avoid Class Parameters and Class Returns 288
Avoid Virtual Functions 289
Avoid Virtual Base Classes. 289
Avoid NULL-Check of Class Pointers 289
Declare near first use. 290
About Code size 290
C++ Overhead. 290
Short C++ Tutorial. 291
1. Classes. 292
Static Members 293
Member Functions 293
Implicit Casting of Classes 293
2. Member Access Control 294
Private 294
Protected 294
Public 294
Example (public, private, protected) 294
Private Base Class 295
Protected Base Class 295
Public Base Class. 295
Example (public, private, protected base class). 296
Friends 296
3. Special Member Functions 297
Constructors. 297
Destructors. 297
Conversion by Constructor 298
Conversion Functions 298
Compiler Generated Functions 299
4. Virtual Functions 299
Access to Virtual Functions. 301
Function Specifiers 302
Single Inheritance and Virtual Functions 302
Multiple Inheritance and Virtual Functions. 303
Virtual Tables Naming Convention. 305
5. Virtual Base Classes 306
Multiple Base Classes 306
Implementation 309
Virtual Base Classes and Casting 311
Multiple Access 311
6. Templates 312
Class Templates. 313
Function Templates 313
Member Function Templates. 313

Template Arguments . . . 314
Static Members and Variables . . . 314
Declarations and Definitions . . . 315
7. Exception Handling . . . 315
8. Pure Virtual Functions / Abstract Classes . . . 315
Pure Virtual Functions . . . 315
Abstract Classes . . . 316
9. Overloading . . . 321
Default Arguments . . . 321
Operator new and delete . . . 321
10. Pointer to Members . . . 323
Pointers to Static Members . . . 326
C++ Name Encoding, Type-safe Linkage . . . 326

Generating Compact Code . . . 331
Compiler Options . . . 331
-Or: Register Optimization . . . 331
-Cni: Non integral promotion on integer . . . 331
-Oi: Inline Functions . . . 331
-Oc: Common Subexpression Elimination . . . 332
SHORT Segments . . . 332
Defining IO Registers . . . 333
Programming Guide Lines . . . 334
HLI Assembly . . . 334
Post/Pre Operators in Complex Expressions . . . 335
Boolean Types . . . 335
Printf/Scanf . . . 336
Bitfields . . . 336
Struct Returns . . . 336
Local Variables . . . 337
Parameter Passing . . . 338
Unsigned Data Types . . . 338
Inlining/Macros . . . 338
abs() and absl() . . . 338
memcpy() and memcpy2() . . . 339
Data Types . . . 339
Short Segments . . . 339
Qualifiers . . . 340

Motorola HC08 Back End . . . 341
Memory Models . . . 341
SMALL Model . . . 341
BANKED Model . . . 341
Non–banked Memory . . . 341
Non-ANSI Keywords . . . 342

Data Types 342
Scalar Types 342
Floating Point Types 343
Bit Fields 344
Pointer Types, Function Pointers 346
Division and Modulus 346
Structured Types, Alignment 347
Object Size 347
Register Usage 347
Call Protocol and Calling Conventions 347
Argument Passing 347
Return Values 348
Returning Large Objects 348
Far and Near Calling Convention 348
Near Calling Convention 348
Specifying the Calling Convention 350
Far call runtime routine 350
Adaptation of the far calling runtime routines to the Hardware 350
Stack usage during a far call 351
Function Pointer 351
Stack Frames 351
Frame Pointer 351
Entry Code 351
Exit Code 352
Pragma TRAP_PROC 352
Interrupt Vector Table Allocation 353
Segmentation 353
The _OVERLAP Segment (Compiler Option -Cv) 354
The _DATA_ZEROPAGE Segment 355
Optimizations 355
Lazy Instruction Selection 356
Strength Reduction 356
Shift Optimizations 356
Accessing Bitfields 356
Branch Optimizations 357
Optimization for Execution Time or Code Size 357
Register Tracing 357
Lazy Mode (Option -Or) 358
#pragma SAFE_OPT_ONLY 358
Volatile Objects, Absolute Variables 359
Optimization of Pointer Additions (Option -Ca) 359
Generating Compact code With the CHC08 Compiler 360
Compiler Options 360
SHORT Segments 361
Defining IO Registers 362

High Level Inline Assembler for Motorola HC08 365
Syntax 365

Inline Assembly Language 365
Examples 366
Special Features 367
Reserved Words 367
Pseudo-Opcodes 367
Accessing Variables 367
Address Notation 368
H:X Instructions 368
Constant Expressions 368
Optimizing Inline Assembly 369
Assertions 369
Stack Adjust 369
In & Gen Sets 370
Entry, Exit 371
Optimization Info 372
Getting the High Address Part in HLI 373

Messages 375
Message Kinds 375
INFORMATION 375
WARNING 375
ERROR 375
FATAL 375
DISABLE 375
Message Details 375
Message List 376
C1: Unknown message occurred 376
C2: Message overflow, skipping <kind> messages 376
C64: Line Continuation occurred in <FileName> 377
C1000: Illegal identifier list in declaration 377
C1001: Multiple const declaration makes no sense 377
C1002: Multiple volatile declaration makes no sense 378
C1003: Illegal combination of qualifiers 378
C1004: Redefinition of storage class 378
C1005: Illegal storage class 378
C1006: Illegal storage class 378
C1007: Type specifier mismatch 378
C1008: Typedef name expected 379
C1009: Invalid redeclaration 379
C1010: Illegal enum redeclaration 379
C1012: Illegal local function definition 379
C1014: Integral type expected or enum value out of range 380
C1015: Type is being defined 380
C1016: Parameter redeclaration not permitted 380
C1017: Empty declaration 380
C1018: Illegal type composition 381
C1019: Incompatible type to previous declaration 381
C1020: Incompatible type to previous declaration 381
C1021: Bit field type is not 'int' 381

C1022: 'far' used in illegal context 382
C1023: 'near' used in illegal context 382
C1024: Illegal bit field width 382
C1025: ',' expected before '...' 382
C1026: Constant must be initialized 382
C1027: Reference must be initialized 383
C1028: Member functions cannot be initialized 383
C1029: Undefined class 383
C1030: Pointer to reference illegal 383
C1031: Reference to reference illegal 383
C1032: Invalid argument expression 384
C1033: Ident should be base class or data member 384
C1034: Unknown kind of linkage 384
C1035: Friend must be declared in class declaration 384
C1036: Static member functions cannot be virtual 385
C1037: Illegal initialization for extern variable in block scope 385
C1038: Cannot be friend of myself 385
C1039: Typedef-name or ClassName expected 385
C1040: No valid :: classname specified 386
C1041: Multiple access specifiers illegal 386
C1042: Multiple virtual declaration makes no sense 386
C1043: Base class already declared in base list 386
C1044: User defined Constructor is required 386
C1045: <Special member function> not generated. 387
C1046: Cannot create compiler generated <Special member function> for nameless class 387
C1047: Local compiler generated <Special member function> not supported. 387
C1048: Generate compiler defined <Special member function> 387
C1049: Members cannot be extern 388
C1050: Friend must be a class or a function 388
C1051: Invalid function body 388
C1052: Unions cannot have class/struct object members containing Con/Destructor/Assign-Operator . . 388
C1053: Nameless class cannot have member functions 389
C1054: Incomplete type or function in class/struct/union 389
C1055: External linkage for class members not possible 389
C1056: Friend specifier is illegal for data declarations 389
C1057: Wrong return type for <FunctionKind> 390
C1058: Return type for FunctionKind must be <ReturnType> 390
C1059: Parameter type for <FunctionKind> parameter <No> must be <Type> 390
C1060: <FunctionKind> wrong number of parameters 390
C1061: Conversion operator must not have return type specified before operator keyword 390
C1062: Delete can only be global, if parameter is (void *) 391
C1063: Global or static-member operators must have a class as first parameter 391
C1064: Constructor must not have return type 391
C1065: 'inline' is the only legal storage class for Constructors. 391
C1066: Destructor must not have return type 392
C1067: Object is missing decl specifiers 392
C1068: Illegal storage class for Destructor 392
C1069: Wrong use of far/near/rom/uni/paged in local scope 392
C1070: Object of incomplete type 392
C1071: Redefined extern to static 393
C1072: Redefined extern to static 393
C1073: Linkage specification contradicts earlier specification 393
C1074: Wrong member function definition 393
C1075: Typedef object id already used as tag 394
C1076: Illegal scope resolution in member declaration 394

C1077: <FunctionKind> must not have parameters. 394
C1078: <FunctionKind> must be a function 394
C1080: Constructor/destructor: Parenthesis missing 395
C1081: Not a static member. 395
C1082: <FunctionKind> must be non-static member of a class/struct 395
C1084: Not a member 395
C1085: <ident> is not a member 396
C1086: Global unary operator must have one parameter. 396
C1087: Static unary operator must have one parameter. 396
C1088: Unary operator must have no parameter 396
C1089: Global binary operator must have two parameters 396
C1090: Static binary operator must have two parameters 397
C1091: Binary operator must have one parameter. 397
C1092: Global unary/binary operator must have one or two parameters. 397
C1093: Static unary/binary operator must have one or two parameters 397
C1094: Unary/binary operator must have no or one parameter 397
C1095: Postfix ++/-- operator must have integer parameter 398
C1096: Illegal index value 398
C1097: Array bounds missing 398
C1098: Modifiers for non-member or static member functions illegal 398
C1099: Not a parameter type 398
C1100: Reference to void illegal 399
C1101: Reference to bitfield illegal 399
C1102: Array of reference illegal. 399
C1103: Second C linkage of overloaded function not allowed 399
C1104: Bit field type is neither integral nor enum type 399
C1105: Backend does not support non-int bitfields 400
C1106: Non-standard bitfield type. 400
C1107: Long long bit fields not supported yet. 400
C1108: Constructor cannot have own class/struct type as first and only parameter 400
C1109: Generate call to Copy Constructor 400
C1110: Inline specifier is illegal for data declarations. 401
C1111: Bitfield cannot have indirection. 401
C1112: Interrupt specifier is illegal for data declaration 401
C1113: Interrupt specifier used twice for same function 401
C1114: Illegal interrupt number. 402
C1115: Template declaration must be class or function 402
C1116: Template class needs a tag. 402
C1117: Illegal template/non-template redeclaration 402
C1118: Only bases and class member functions can be virtual 402
C1119: Pure virtual function qualifier should be (=0) 403
C1120: Only virtual functions can be pure. 403
C1121: Definition needed if called with explicit scope resolution 403
C1122: Cannot instantiate abstract class object 404
C1123: Cannot instantiate abstract class as argument type 404
C1124: Cannot instantiate abstract class as return type 405
C1125: Cannot instantiate abstract class as a type of explicit conversion 405
C1126: Abstract class cause inheriting pure virtual without overriding function(s) 406
C1127: Constant void type probably makes no sense 406
C1128: Class contains private members only 406
C1129: Parameter list missing in pointer to member function type. 407
C1130: This C++ feature is disabled in your current cC++/EC++ configuration 407
C1131: Illegal use of global variable address modifier 407
C1132: Cannot define an anonymous type inside parentheses 408
C1390: Implicit virtual function. 408
C1391: Pseudo Base Class is added to this class 409

C1392: Pointer to virtual methods table not qualified for code address space (use -Qvtprom or -Qvtpuni) . . 410
C1393: Delta value does not fit into range (option -Tvtd) 410
C1395: Classes should be the same or derive one from another. 411
C1396: No pointer to STATIC member: use classic pointer 412
C1397: Kind of member and kind of pointer to member are not compatible 412
C1400: Missing parameter name in function head 413
C1401: This C++ feature is not yet implemented 413
C1402: This C++ feature is not yet implemented 413
C1403: Out of memory 414
C1404: Return expected 414
C1405: Goto <undeclared Label> in this function 414
C1406: Illegal use of identifierList 414
C1407: Illegal function-redefinition 414
C1408: Incorrect function-definition 415
C1409: Illegal combination of parameterlist and identlist 415
C1410: Parameter-declaration - identifier-list mismatch 415
C1411: Function-definition incompatible to previous declaration 415
C1412: Not a function call, address of a function. 415
C1413: Illegal label-redeclaration. 416
C1414: Casting to pointer of non base class 416
C1415: Type expected. 416
C1416: No initializer can be specified for arrays 416
C1417: Const/volatile not allowed for type of new operator 416
C1418:] expected for array delete operator 417
C1419: Non-constant pointer expected for delete operator 417
C1420: Result of function-call is ignored 417
C1421: Undefined class/struct/union 417
C1422: No default Ctor available 418
C1423: Constant member must be in initializer list 418
C1424: Cannot specify explicit initializer for arrays 418
C1425: No Destructor available to call. 419
C1426: Explicit Destructor call not allowed here 419
C1427: 'this' allowed in member functions only 419
C1428: No wide characters supported 419
C1429: Not a destructor id 419
C1430: No destructor in class/struct declaration 420
C1431: Wrong destructor call 420
C1432: No valid classname specified 420
C1433: Explicit Constructor call not allowed here 420
C1434: This C++ feature is not yet implemented 421
C1435: Return expected 421
C1436: delete needs number of elements of array 421
C1437: Member address expected 422
C1438: ... is not a pointer to member ident 423
C1800: Implicit parameter-declaration (missing prototype). 423
C1801: Implicit parameter-declaration 424
C1802: Must be static member 424
C1803: Illegal use of address of function compiled under the pragma REG_PROTOTYPE 424
C1804: Ident expected. 424
C1805: Non standard conversion used 424
C1806: Illegal cast-operation 425
C1807: No conversion to non-base class 425
C1808: Too many nested switch-statements. 425
C1809: Integer value for switch-expression expected 426

C1810: Label outside of switch-statement . . . 426
C1811: Default-label twice defined . . . 426
C1812: Case-label-value already present . . . 426
C1813: Division by zero. . . . 426
C1814: Arithmetic or pointer-expression expected . . . 427
C1815: <Name> not declared (or typename) . . . 427
C1816: Unknown struct- or union-member . . . 427
C1817: Parameter cannot be converted to non-constant reference . . . 427
C1819: Constructor call with wrong number of arguments . . . 428
C1820: Destructor call must have 'void' formal parameter list . . . 428
C1821: Wrong number of arguments . . . 428
C1822: Type mismatch. . . . 429
C1823: Undefining an implicit parameter-declaration. . . . 429
C1824: Indirection to different types . . . 429
C1825: Indirection to different types . . . 429
C1826: Integer-expression expected . . . 430
C1827: Arithmetic types expected . . . 430
C1828: Illegal pointer-subtraction . . . 430
C1829: + - incompatible Types . . . 430
C1830: Modifiable lvalue expected . . . 431
C1831: Wrong type or not an lvalue . . . 431
C1832: Const object cannot get incremented. . . . 431
C1833: Cannot take address of this object . . . 431
C1834: Indirection applied to non-pointer . . . 431
C1835: Arithmetic operand expected. . . . 432
C1836: Integer-operand expected. . . . 432
C1837: Arithmetic type or pointer expected . . . 432
C1838: Unknown object-size: sizeof(incomplete type) . . . 432
C1839: Variable of type struct or union expected . . . 432
C1840: Pointer to struct or union expected . . . 433
C1842: [incompatible types. . . . 433
C1843: Switch-expression: integer required . . . 433
C1844: Call-operator applied to non-function . . . 433
C1845: Constant integer-value expected . . . 433
C1846: Continue outside of iteration-statement. . . . 434
C1847: Break outside of switch or iteration-statement . . . 434
C1848: Return <expression> expected. . . . 434
C1849: Result returned in void-result-function . . . 434
C1850: Incompatible pointer operands . . . 434
C1851: Incompatible types. . . . 435
C1852: Illegal sizeof operand. . . . 435
C1853: Unary minus operator applied to unsigned type . . . 435
C1854: Returning address of local variable . . . 435
C1855: Recursive function call . . . 436
C2000: No constructor available . . . 436
C2001: Illegal type assigned to reference. . . . 436
C2004: Non-volatile reference initialization with volatile illegal . . . 436
C2005: Non-constant reference initialization with constant illegal . . . 436
C2006: (un)signed char reference must be const for init with char . . . 437
C2007: Cannot create temporary for reference in class/struct . . . 437
C2008: Too many arguments for member initialization . . . 437
C2009: No call target found! . . . 437
C2010: <Name> is ambiguous. . . . 438
C2011: <Name> can not be accessed. . . . 438
C2012: Only exact match allowed yet or ambiguous! . . . 439
C2013: No access to special member of base class . . . 439

C2014: No access to special member of member class 440
C2015: Template is used with the wrong number of arguments 440
C2016: Wrong type of template argument 440
C2017: Use of incomplete template class 441
C2018: Generate class/struct from template 441
C2019: Generate function from template 441
C2020: Template parameter not used in function parameter list 441
C2021: Generate NULL-check for class pointer 442
C2022: Pure virtual can be called only using explicit scope resolution 442
C2023: Missing default parameter 443
C2024: Overloaded operators cannot have default arguments 443
C2025: Default argument expression can only contain static or global objects or constants 443
C2200: Reference object type must be const 443
C2201: Initializers have too many dimensions 444
C2202: Too many initializers for global Ctor arguments 444
C2203: Too many initializers for Ctor arguments 444
C2204: Illegal reinitialization 445
C2205: Incomplete struct/union, object can not be initialized 445
C2206: Illegal initialization of aggregate type 445
C2207: Initializer must be constant. 446
C2209: Illegal reference initialization. 446
C2210: Illegal initialization of non-aggregate type 446
C2211: Initialization of a function 447
C2212: Global variable initialization is not supported yet 447
C2401: Pragma <ident> expected 447
C2402: Variable <Ident> <State> 447
C2450: Expected: <list of expected keywords and tokens> 447
C2550: Too many nested scopes. 448
C2700: Too many numbers 448
C2701: Illegal floating-point number 448
C2702: Number too large for float 449
C2703: Illegal character in float number 449
C2704: Illegal number. 449
C2705: Possible loss of data 449
C2706: Octal Number 450
C2800: Illegal operator 450
C2801: <Symbol> missing" 450
C2802: Illegal character found: <Character> 450
C2803: Limitation: Parser was going out of sync! 451
C2900: Constant condition found, removing loop 451
C2901: Unrolling loop 451
C3000: File-stack-overflow (recursive include?) 451
C3100: Flag stack overflow -- flag ignored 452
C3200: Source-file too big 452
C3300: String-buffer overflow 452
C3301: Catenated string too long 452
C3302: Prae_number-buffer overflow 453
C3303: Implicit concatenation of strings 453
C3304: Too many internal ids, split up compilation unit 453
C3400: Cannot initialize object (dest too small). 453
C3500: Not supported fixup-type for ELF-Output occurred 454
C3600: Function has no code: remove it! 454
C3601: Pragma TEST_CODE: mode <Mode>, size given <Size> expected <Size>, hashcode given <HashCode>, expected <HashCode>454
C3602: Global objects: <Number>, Data Size (RAM): <Size>, Const Data Size (ROM): <Size> 454

C3603: Static ‘<Function>’ was not defined 455
C3604: Static ‘<Object>’ was not referenced 455
C3605: Runtime object '<Object>' is used at PC <PC> 455
C3606: Initializing object '<Object>' 456
C3700: Special opcode too large 456
C3701: Too many attributes for DWARF2.0 Output 456
C3800: Segment name already used. 456
C3801: Segment already used with different attributes 456
C3802: Segment pragma incorrect 457
C3803: Illegal Segment Attribute. 457
C3900: Return value too large 457
C4000: Condition always is TRUE 457
C4001: Condition always is FALSE 458
C4002: Result not used. 458
C4003: Shift count converted to unsigned char 458
C4004: BitSet/BitClr bit number converted to unsigned char 458
C4005: Expression with a cast is not a lvalue 459
C4006: Expression too complex. 459
C4100: Converted bit field signed -1 to 1 in comparison 459
C4101: Address of bitfield is illegal. 460
C4200: Other segment than in previous declaration 460
C4201: pragma <name> was not handled 460
C4202: invalid pragma OPTION, <description> 461
C4203: invalid pragma MESSAGE, <description> 461
C4300: Call of an empty function removed 461
C4301: Inline expansion done for function call 462
C4302: Could not generate inline expansion for this function call 462
C4400: Comment not closed 462
C4401: Recursive comments not allowed 462
C4402: Redefinition of existing macro ‘<MacroName>’ 463
C4403: Macro-buffer overflow 463
C4404: Macro parents not closed. 463
C4405: Include-directive followed by illegal symbol 463
C4406: Closing '>' missing. 464
C4407: Illegal character in string or closing '>' missing 464
C4408: Filename too long 464
C4443: Undefined Macro 'MacroName' is taken as 0 464
C4700: Illegal pragma TEST_ERROR 465
C4701: pragma TEST_ERROR: Error <ErrorNumber> did not occur 465
C4800: Implicit cast in assignment 465
C4801: Too many initializers. 465
C4802: String-initializer too large 466
C4900: Function differs in return type only. 466
C5000: Following condition fails: sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)466
C5001: Following condition fails: sizeof(float) <= sizeof(double) <= sizeof(long double) <= sizeof(long long double)466
C5002: Illegal type 466
C5003: Unknown array-size. 467
C5004: Unknown struct-union-size 467
C5005: PACE illegal type 467
C5006: Illegal type settings for HI-CROSS Object File Format 467
C5100: Code size too large. 468
C5200: 'FileName' file not found 468
C5300: Limitation: code size > 'Size' bytes 468
C5301: Limitation: too many procedures (>'Number') 468

C5700: Internal Error <ErrorNumber> in '<Module>', please report to <Producer>. 468
C5701: Internal Error #<ErrorNumber> in '<Module>' while compiling file '<File>, procedure '<Function>', please report to <Producer>469
C5702: <Variable>: local variable declared in function <Function> but not referenced 469
C5703: <Parameter>: parameter declared in function <Function> but not referenced 469
C5800: User requested stop 470
C5900: Result is zero 470
C5901: Result is one 470
C5902: Shift count is zero. 470
C5903: Zero modulus 470
C5904: Division by one 471
C5905: Multiplication with one 471
C5906: Subtraction with zero 471
C5907: Addition replaced with shift 471
C5908: Constant switch expression 472
C5909: Assignment in condition. 472
C5910: Label removed 472
C5911: Division by zero at runtime 472
C5912: Code in 'if' and 'else' part are the same 473
C5913: Conditions of ’if’ and ’else if’ are the same 473
C5914: Conditions of ’if’ and ’else if’ are inverted 473
C5915: Nested 'if' with same conditions 474
C5916: Nested 'if' with inverse conditions 474
C6000: Creating Asm Include File <file>. 474
C6001: Could not Open Asm Include File because of <reason> 475
C6002: Illegal pragma CREATE_ASM_LISTING because of <reason> 475

ANSI Library Reference. 477
Library Files 477
Directory Structure 477
Common Source Files. 478
Startup Files 478
Library Files 478
Special Features 479
Memory Management - malloc, free, calloc, realloc; alloc.c, heap.c 479
Signals - signal.c 479
Multi-byte Characters - mblen, mbtowc, wctomb, mbstowcs, wcstombs; stdlib.c. 480
Program Termination - abort, exit, atexit; stdlib.c. 480
I/O - printf.c 480
Locales - locale.* 482
ctype 482
Library Structure 482
Error Handling 482
String Handling Functions 482
Memory Block Functions 483
Mathematical Functions 483
Memory Management 484
Searching and Sorting 485

Character Functions 485
System Functions 486
Time Functions. 486
Locale Functions 487
Conversion Functions. 487
printf and scanf. 487
File I/O 487
Types and Macros in the Standard Library 488
errno.h 489
float.h. 489
limits.h. 489
locale.h 490
math.h 491
setjmp.h 492
signal.h 492
stddef.h 492
stdio.h 492
stdlib.h 493
time.h. 493
string.h. 494
assert.h. 494
stdarg.h 494
ctype.h 495
The Standard Functions 496
abort. 496
abs 497
acos, acosf 497
asctime. 497
asin, asinf. 497
assert 498
atan, atanf 498
atan2, atan2f 498
atexit 499
atof. 499
atoi 499
atol 500
bsearch. 500
calloc 501
ceil, ceilf 501
clearerr. 501
clock 502
cos, cosf. 502
cosh, coshf. 502
ctime 502
difftime 503
div 503

exit. 503
exp, expf 503
fabs, fabsf 504
fclose 504
feof 504
ferror 504
fflush 505
fgetc. 505
fgetpos. 505
fgets. 506
floor, floorf 506
fmod, fmodf 506
fopen 507
fprintf 508
fputc 508
fputs. 508
fread 508
free 508
freopen 509
frexp, frexpf 509
fscanf. 509
fseek 509
fsetpos 510
ftell 510
fwrite 511
getc 511
getchar. 511
getenv 511
gets 511
gmtime 512
isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit 512
labs 513
ldexp, ldexpf 513
ldiv 513
localeconv 514
localtime 514
log, logf 514
log10, log10f. 514
longjmp 515
malloc 515
mblen. 515
mbstowcs. 515
mbtowc 516
memchr 516
memcmp 516
memcpy, memmove 516

memset . . . 517
mktime . . . 517
modf, modff . . . 517
perror . . . 517
pow, powf . . . 518
printf . . . 518
putc . . . 518
putchar . . . 518
puts . . . 519
qsort . . . 519
raise . . . 520
rand . . . 520
realloc . . . 520
remove . . . 520
rename . . . 521
rewind . . . 521
scanf . . . 521
setbuf . . . 521
setjmp . . . 522
setlocale . . . 522
setvbuf . . . 523
signal . . . 523
sin, sinf . . . 524
sinh, sinhf . . . 524
sprintf . . . 524
sqrt, sqrtf . . . 528
srand . . . 528
sscanf . . . 528
strcat . . . 531
strchr . . . 531
strcmp . . . 532
strcoll . . . 532
strcpy . . . 532
strcspn . . . 532
strerror . . . 533
strftime . . . 533
strlen . . . 534
strncat . . . 534
strncmp . . . 535
strncpy . . . 535
strpbrk . . . 535
strrchr . . . 535
strspn . . . 535
strstr . . . 536
strtod . . . 536
strtok . . . 536
strtol . . . 537
strtoul . . . 537

strxfrm 538
system 538
tan, tanf 538
tanh, tanhf 538
time 539
tmpfile 539
tmpnam 539
tolower 539
toupper 540
ungetc 540
va_arg, va_end, va_start 540
vfprintf, vprintf, vsprintf 541
wctomb 541
wcstombs 541

compactC++ library 543
Introduction 543
Implementation 543
I/O library 545
Description 545
Headers 545
Classes 545
Example 546
String library 546
Description 546
Headers 547
Classes 547
Example 547
Memory library 547
Description 547
Headers 547
Classes 547
Example 548
Complex library 548
Description 548
Headers 548
Classes 548
Example 548
Bitset Library 549
Description: 549
Headers: 549
Classes: 549
Example: 549
Strstream Library 550

Description: . . . 550
Headers: . . . 550
Classes: . . . 550
Example: . . . 550
STL (Standard Template Library) . . . 550
Description: . . . 550
Headers: . . . 550

Appendix . . . 553
Migration Hints . . . 553
Allocation of Bit Fields . . . 553
Type Sizes, Sign of Character . . . 553
@tiny Qualifier for Variables . . . 554
Interrupt Function, @interrupt . . . 554
Recursive Comments . . . 554
Frequently Asked Questions (FAQ's), Trouble Shooting . . . 555
Installation . . . 555
Running Programs . . . 555
Making Applications . . . 556
The Compiler Reports an Error, but WinEdit Doesn't Display it . . . 556
Some Programs Can't Find a File . . . 556
The Compiler Seems to Generate Incorrect Code . . . 556
The Code seems to be Correct, but the Application does not work . . . 557
The Linker can't Handle an Object File . . . 557
The Make Utility doesn't Make all of the Application . . . 557
The Make Utility Unnecessarily Re–compiles a File . . . 557
The help file cannot be opened by double clicking on it in the file manager or in the explorer . . . 557
Bug Reports . . . 558
Information . . . 558
Bug . . . 558
Critical Bug . . . 558
Electronic Mail (email) or Fax Report Form . . . 559
Technical Support . . . 560
EBNF Notation . . . 561
EBNF Example . . . 561
Terminal Symbols . . . 561
Non-terminal Symbols . . . 562
Vertical Bar . . . 562
Brackets . . . 562
Parentheses . . . 562
Production End . . . 562
EBNF-Syntax . . . 563
Extensions . . . 563
Abbreviations, Lexical Conventions . . . 564
Number Formats . . . 564
Precedence and Associativity of Operators for ANSI C . . . 565

List of all Escape Sequences 566
Index 569

Copyright and Trademarks

Copyright

This work is subject to copyright. No part of this publication may be reproduced or transmitted in any form or by any means – graphic, electronic, electrical, mechanical, chemical (including photocopying), recording in any medium, etc. without prior written consent from HIWARE.

Disclaimer

While every effort has been made to ensure the accuracy of all information in this document, HIWARE assumes no liability whatsoever to any party for any loss or damage caused by errors or statements of any kind in this document, its updates, supplements or special editions, whether such errors are omissions or incorrect statements due to negligence, accident or any other cause.

HIWARE further assumes no liability whatsoever arising from the application or use of any product described herein; nor any liability for incidental or consequential damages arising from the use of this document or any product described herein.

HIWARE also disclaims any warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness of the product for a particular purpose.

Trademarks

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Globetrotter Software, Inc.

HP and HP-UX are trademarks of Hewlett-Packard Co.

Intel is a trademark of Intel Corporation.

Motorola is a registered trademark of Motorola, Inc.

MS-DOS and Windows are registered trademarks of Microsoft Corporation.

SUN is a trademark of Sun Microsystems, Inc.

UNIX is a registered trademark of X/Open Company, Ltd.

VAX and VMS are trademarks of Digital Equipment Corporation.

All other trademarks are property of their respective owners.

Introduction

This document describes the optimizing HI-CROSS+ ANSI-C/C++ Compiler. The Compiler mainly consist of a **Front End** which is language dependent and a **Back End** which depends on the target processor.

Application Programs

After installation, you will find all your executable programs in the 'prog' subdirectory, e.g. if you installed the software in 'C:\HIWARE' on a PC, you will find all the application programs in 'C:\HIWARE\PROG' (for details see installation guide).

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

```
chc08.exe      Motorola HC05 Compiler
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 for automatically rebuild
burner.exe     Batch and interactive Burner (S-Records, ...)
hiwave.exe     Multi-Purpose Debugging Environment
hitools.exe    Shell and Launching Tool
```

Note that depending on your license configuration not all programs above may be installed or there may be additional programs!

Literature

For further details about programming languages we recommend following publications:

- "American National Standard for Programming Languages – C", ANSI/ISO 9899–1990 (see also http//www.x3.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

This text cannot be a real introduction to the C++ language, programmers not familiar with C++ should read additional documentation. Beside the documents of the ANSI standardization group, we recommend the following books:

- “C++ Programming Guidelines”, Thomas Plum, Dan Saks, Plum Hall Inc., 1989, ISBN 0-911537-07-4
- "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

Highlights

- Powerful User Interface
- On-line Help
- Flexible Type Management
- Flexible Message Management
- 32bit Application
- 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**: FAQ's, Trouble shooting, Technical Notes
- **Index**

Specific Configuration Markers

Some parts of this document are marked with a special box in the upper right corner. This box indicates that this part is only available for a specific configuration. Please check your license configuration.

Box	Description
C++	This section is only available for Compiler with the C++ feature.
+	This marker indicates sections which are valid for HI-CROSS+ Compilers only.
HC08	Only available for Motorola HC08.
PC	Only available for the IBM-PC operating system.
UNIX	Only available for the UNIX operating system.

HI-CROSS+ vs. Smile~Line

+

The following table gives a short overview over the differences in the Compilers of

the two product lines HI-CROSS+ and Smile~Line:

Compiler	HI-CROSS+	Smile~Line
Graphical User Interface	yes	no
On-line Help	yes	no
16bit Application	no	yes
32bit Application	yes	no
C++, EC++, compactC++ available	yes	no
Flexible Type Management	yes	no
Flexible Message Management	yes	no
Inlining	yes	no
CSE Optimization	yes	no
HIWARE Object File Format	yes	yes
ELF/DWARF Object File Format	yes	no
ANSI Library Sources	yes	no
Hardlock	no	yes
FLEXlm	yes	no

C++, EC++, compactC++

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

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

The solution is to provide a compactC++ and EC++, which is are subsets of the hole C++ language, adapted for embedded application programming. This subset of the C++ language avoids implicit and explicit overhead of the C++ language (e.g. virtual member functions, multiple inheritance). The EC++ is a restricted subset, where the cC++ (compact C++) allows to include features which are not in the EC++ definition, thus it is much more flexible.

Another key aspect of compactC++ is a flexible configuration of the language (e.g. allowed keywords, code generation behavior, message management), thus the Compiler can be adapted for the special needs for embedded programming.

Object File Formats

+

The HI-CROSS+ 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), of the library files (extension .lib) and absolute files (extension .abs).

Note: Be careful and do not mix object file formats. Both the HIWARE and the ELF/DWARF object files are using the same extensions and are installed in the same library directory during installation.

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 dramatically smaller than the ELF/DWARF object files and thus allows faster file operations on the object files. The object file format is also relatively easy to support by other tool vendors (e.g. emulators from Abatron, Lauterbach or iSYSTEM) The object file format supports ANSI-C and Modula-2, C++ is supported only with restrictions.

On the other side each other tool vendor has to support this object file format explicitly, and 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 results my also result in slower source/debug info loading: In HIWARE object file format the source position information is given as position information (offset in file), and not directly in file/line/column format, thus a debugger normally has to translate this HIWARE object file format source information format into file/line/column which usually slows down the source file/debugging info loading.

ELF/DWARF Object File Format

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

MOTOROLA itself defined this object file format as the standard for tool vendors supporting MOTOROLA devices. This standard allows inter-tool operability, e.g. it is possible to use the compiler from one tool vendor and the linker from another

one. Thus the customer has the choice to select the best tool in the hole tool chain. Additionally, other third parties (e.g. emulator vendors) only have to support this object file to support a wide range of tool vendors.

Object file sizes are huge compared with the HIWARE object file format which results in more file I/O to generate those object files.

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

Tools

Compiler

The Smile~Line Compiler supports only the HIWARE object file format.

The HI-CROSS+ Compiler supports both object file formats, the default is HIWARE object file format (Compiler option -Fh, -F1 or -F2 to switch to ELF/DWARF).

Decoder

Using the HIWARE object file format, the Decoder named ‘decoder’ has to be used.

Using ELF/DWARF object file format, the Decoder named ‘elfdecod’ has to be used.

Linker

Using the HIWARE object file format, the Linker named ‘linker’ has to be used.

Using ELF/DWARF object file format, the Linker named ‘elflink’ has to be used.

Simulator/Debugger

The HIWAVE Simulator/Debugger supports both object file formats.

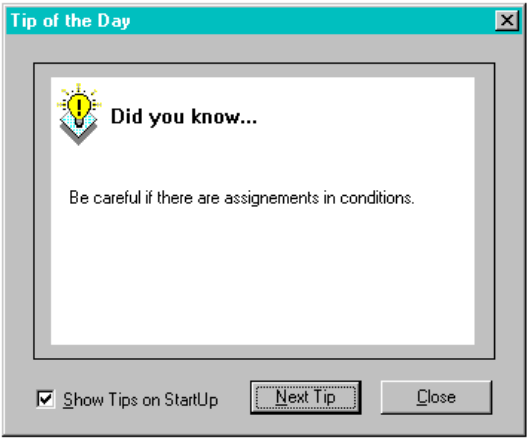
Mixing Object File Formats

Mixing HIWARE and ELF object files is not possible. Mixing ELF object files with DWARF 1.1 and DWARF 2.0 debug info is possible, but the final generated application will not contain any debug info.

HI-CROSS+ Graphical User Interface

+

When you start the Compiler, a standard *Tip of the Day* window is opened containing the last news about the Compiler.



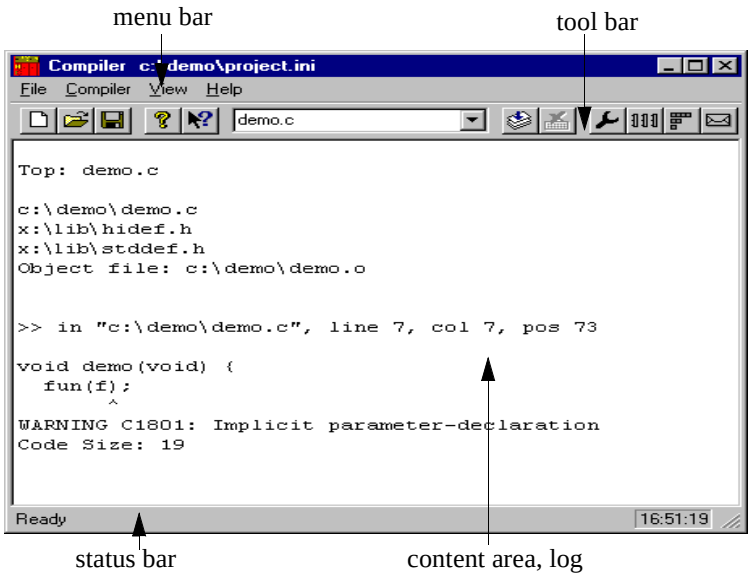
The *Next Tip* button allows you to see the next tip about the Compiler.

If you do not want to open automatically the standard *Tip of the Day* window when the Compiler is started, just unchecked the check box *Show Tips on StartUp*. Note that this configuration entry is stored in the local project file.

If you want to enable automatic display from the standard *Tip of the Day* window again when the Compiler is started, just select the entry *Help* | *Tip of the Day*.... The window will be opened and you can 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 you do not specify any file name while starting the Compiler.

The Compiler 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 compiler name and also the project name. If currently no project is loaded, “Default Configuration” is displayed. A “*” after the configuration name indicates if some values have been changed. Note that not only changes of options, but also the editor configuration and the appearance can make the “*” appear.

Content Area

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

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

When a file is dropped into the Compiler window content area, the corresponding file is either loaded as configuration or compiled. It is loaded as configuration if the file has the extension “ini”. If not, the file is compiled with the current option settings.

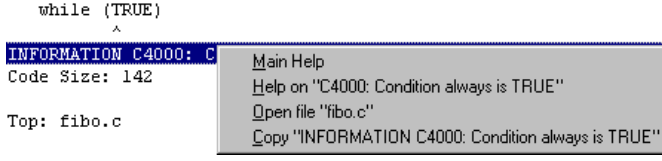
All text in the Compiler window content area can have context information. The context information consists of two items :

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

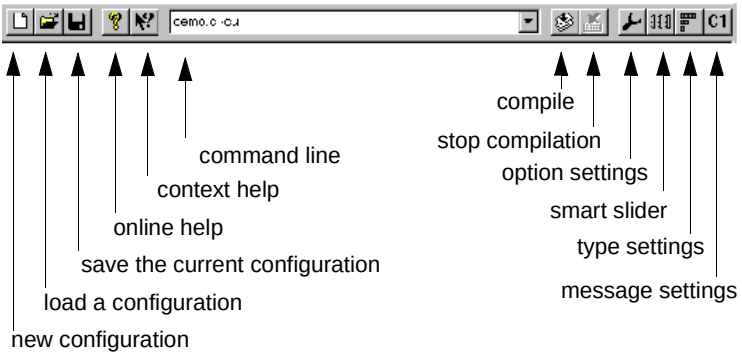
File context information is available for all output where a text file is considered. Especially it is available for all source and include files and for messages, which do concern a specific file. If a file context is available, double-clicking on the text or message opens this file in a editor as specified in the editor configuration. Also with the right mouse button a context menu can be opened. It contains a “Open ..” entry if a file context is available. Note that under Win32s no context menu is available. If a file can not be opened although a context menu entry is present, see the section Editor Configuration below.

The message number is available for any message output. Then 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 only available if a message number is available. The context menu is not available under Win32s.

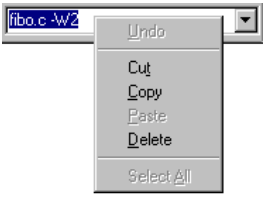


Tool Bar



The three buttons on the left are linked with the corresponding entries of the menu *File*. The next button opens the about dialog. After pressing the context help button (or the shortcut Shift F1), the mouse cursor changes it's form and has now an question mark beside the arrow. Then the help is called for the next item which is clicked at. It can be clicked on menus, toolbar buttons and on the window area to get help specific for this topic.

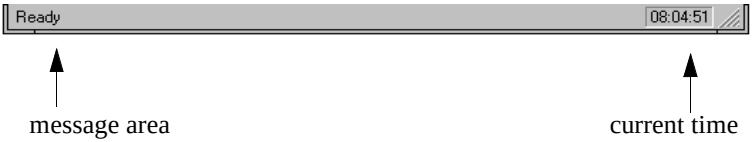
The command line history contains a list of the commands executed. Once you have selected or entered a command in history, clicking *Compile* will start execution of the command. Additionally there is a context menu associated with the command line:



The *Stop Compilation* button allows you to stop the current compilation session.

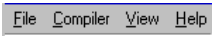
The last four buttons allow the opening of option setting, the smart slider, the type setting and the message setting dialog. Under Win32s not all dialogs are available.

Status Bar



When pointing to a button in the tool bar or a menu entry, the message area will display the function of the button or menu entry you are pointing to.

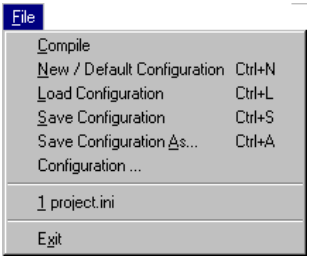
Menu Bar



Following menus are available in the menu bar:

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

File Menu



With the File Menu Compiler configuration files can be saved or loaded. A Compiler configuration file contains following information:

- the Compiler option settings specified in the Compiler dialog boxes

- the Message settings which specify which messages to display and which to treat as error.
- the list of the last command line executed and the current command line.
- 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 have standard extension `.ini`. The user can define as many configuration files as required for his project, and can switch between the different configuration files using the *File | Load Configuration* and *File | Save Configuration* menu entry or the corresponding tool bar buttons.

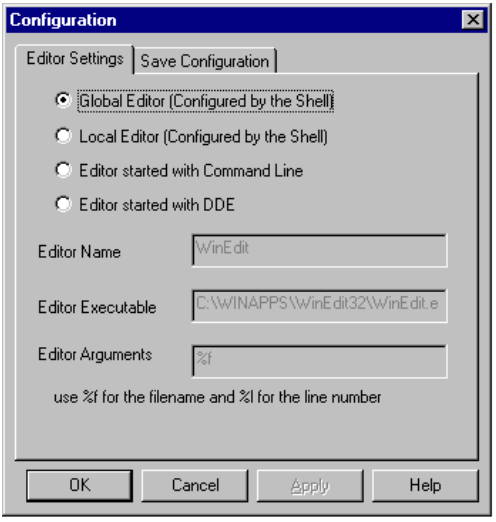
Menu entry	**Description**
Compile	Opens a standard Open File box. The selected file will be compiled as soon as the open File box is closed using *OK*.
New/Default Configuration	Resets the Compiler option settings to the default value. The Compiler options which are activated per default are specified in section *Command Line Options* from 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 compilation session.
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 *Configuration* 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 can be accessed to open a recently opened project again.
Exit	Closes the Compiler.

Edit Settings Dialog

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

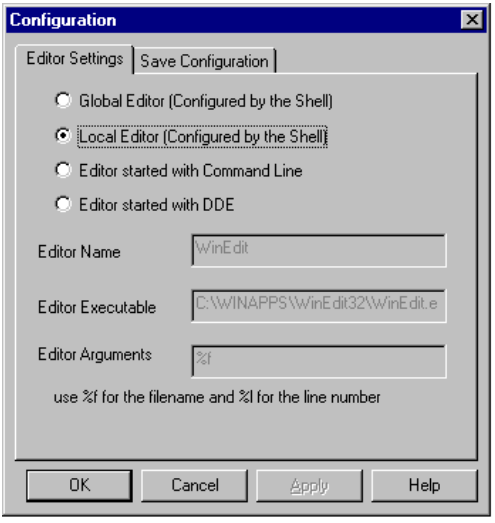
There are the following main entries :

- Global Editor (Configured by the Shell)



This entry is only enabled (black) when in the global initialization file "MCU-TOOLS.INI" a configured editor exists in the "[Editor]" section.

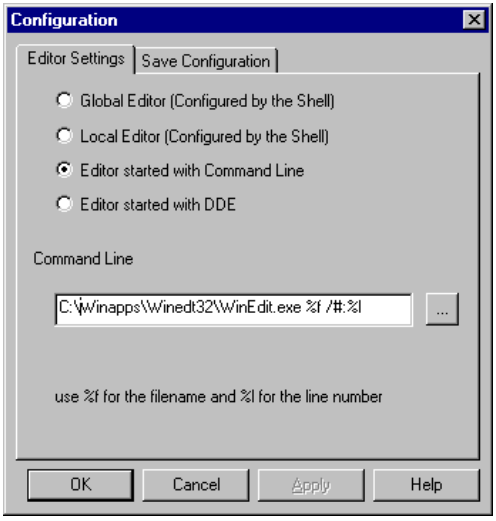
- Local Editor (Configured by the Shell)



This entry is only enabled (black) when in the local initialization file, usually "project.ini" in the current directory, a configured editor exists in the "[Editor]" section.

The Global and Local Editor configuration can not be edited with the compiler, it can only be read. These entries can be configured with the Shell (see Manual for the Shell Tool).

- Editor started with Command Line



Enter the command, which should be used to start the editor. See the note below for modifiers for file name and line number.

The format from the editor command depends on the syntax which should be used to start the editor.

Example: (also look at the notes below)

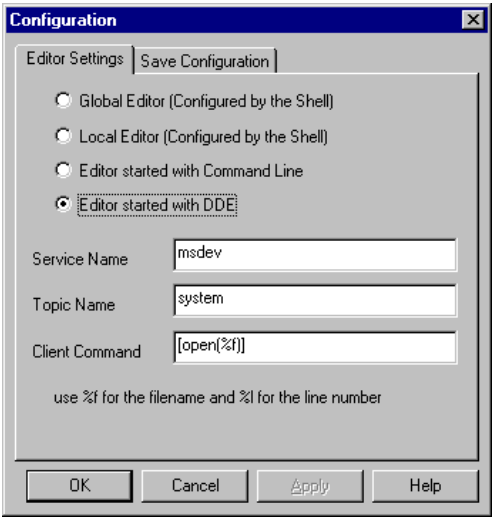
For Winedit 32 bit version use (with an adapted path to the winedit.exe file)

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

For Write.exe use (with an adapted path to the write.exe file, note that write does not support line number).

```
C:\Winnt\System32\Write.exe %f
```

- Editor started with DDE



Enter the service, topic and client name to be used for a DDE connection to the editor. All entries can have modifiers for file name and line number as explained below.

Example:

For Microsoft Developer Studio use the following setting :

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

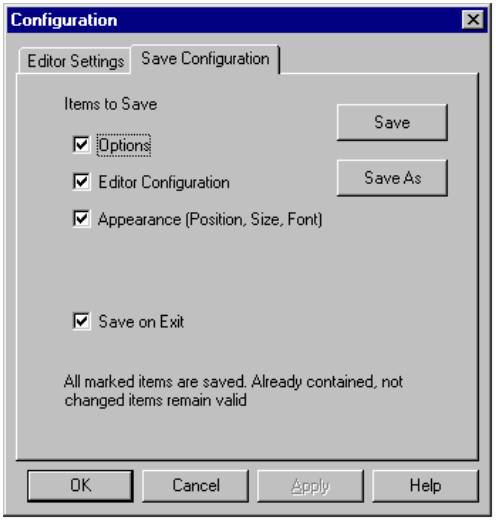
The configurations should contain some modifiers to tell the editor which file to open and at which line.

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

Note: Be careful, the %l modifier can only be used with an editor which can be started with a line number as parameter. This is not the case for WinEdit version 3.1 or lower or for the Notepad. When you work with such an editor, you can 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 your editor manual to define the command line which should be used to start the editor.

Save Configuration Dialog



On the second page of the configuration dialog all options considering the save operation are contained.

In the save configuration dialog, the parts which should be stored into a project file can be configured.

This dialog has the following configurations :

- Options: Only when this mark is set, the current option and message setting are contained when a configuration is written. By disabling this options, the last saved content remains valid.
- Editor Configuration: Only when this mark is set, the current editor setting are contained when a configuration is written. By disabling this options, the last saved content remains valid.
- Appearance: This save topics consists of many parts as the window position (only loaded at startup time) and the command line content and history. Only when this mark is set, these settings are contained when a configuration is written.

Note: By disabling selective options only some parts of a configuration file can be

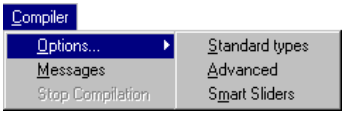
written. For example when the best compiler options are found, the save option mark can be removed. Then future save commands will not modify the options any more.

- Save on Exit: If this option is set, the compiler will write the configuration on exit. No question will appear to confirm this operation. If this option is not set, the compiler will not write the configuration at exit, even if options or another part of the configuration has changed. No confirmation will appear in any case when closing the compiler.

Note: Almost all settings are stored in the configuration file only.
The only exceptions are :
- The recently used configuration list.
- All settings in this dialog.

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

Compiler Menu

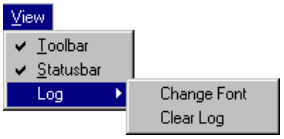


This menu allows you to customize the Compiler. You can graphically set or reset Compiler options or define the optimization level you want to reach.

Menu entry	Description
Options...	allows you to customize the Compiler. You can graphically set/reset Compiler options. Following entries are available when *Options...* is selected:
Standard Types	allows you to specify the size you want to associate with each ANSI C standard type (See *Standard Types Dialog Box* below).
Advanced	allows you to define the options which must be activated when compiling an input file (See *Advanced Options Dialog Box* below).

Menu entry	Description
Smart Slider	allows you to define the optimization level you want to reach, when compiling the input file (See *Compiler Smart Control Dialog Box* below).
Messages	opens a dialog box, where the different error, warning or information messages can be mapped to another message class (See *Message Setting Dialog Box* below).
Stop Compilation	stops immediately the current compilation session.

View Menu



This menu allows you to customize the Compiler window. You can define if the status bar or the tool bar must be displayed or hidden. You can also define the font used in the window or clear the window.

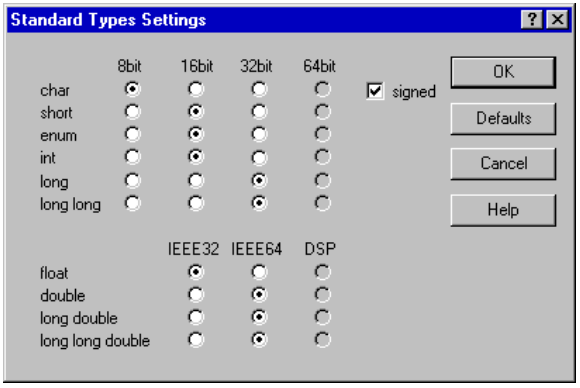
Menu entry	Description
Tool Bar	switches display from the tool bar in the Compiler window.
Status Bar	switches display from the status bar in the Compiler window.
Log...	allows you to customize the output in the Compiler window content area. Following entries are available when *Log...* is selected:
Change Font	opens a standard font selection box. The options selected in the font dialog box are applied to the Compiler window content area.
Clear Log	allows you to clear the Compiler window content area.

Help Menu

This menu allows you to customize if the Tip of the Day Dialog is shown during startup of the Compiler. Additionally it provides a standard Windows Help entry and an entry to an About box.

Menu entry	Description
Tip of the Day	Switches on or off the display of a Tip of the Day during startup.
Help Topics	Standard Help topics.
About ...	Displays an About box with some version and license information.

Standard Types Dialog Box



This dialog box allows you to define the size you want to associate to each ANSI C standard type.

Note: Not all formats may be available for a target. Additionally there has to be at least one type for each size, e.g. it is illegal to specify all types to a size of

32bits, because there is no type for 8bit and 16bit available for the Compiler. Be careful if the HIWARE Object File Format is used instead the ELF/DWARF Object File Format: the HIWARE Format does not support a size > 1 for the char type.

Be careful, following rules must still apply when you modify the size associated with an ANSI C standard type:

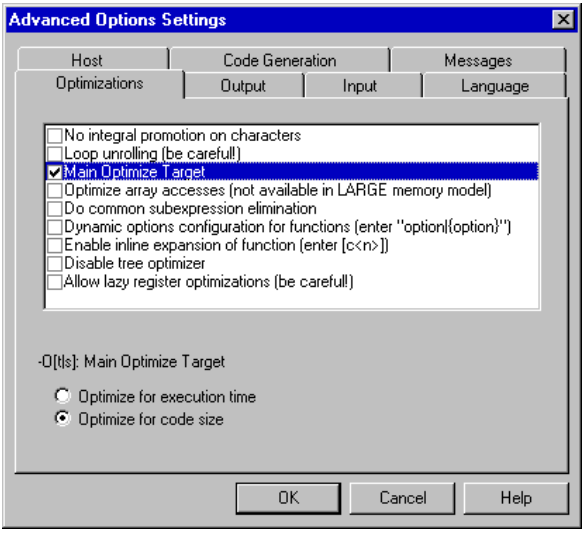
```
sizeof(char)   <= sizeof(short)
sizeof(short)  <= sizeof(int)
sizeof(int)    <= sizeof(long)
sizeof(long)   <= sizeof(long long)
sizeof(float)  <= sizeof(double)
sizeof(double)<= sizeof(long double)
```

Enumerations has to be smaller or equal than ‘int’.

The check box *signed* allows 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.

Advanced Options Dialog Box



This dialog box allows you to set/reset Compiler options. The possible command

line option is also displayed in the lower display area (e.g. -Os/-Ot) The options available are arranged into different groups, and a sheet is available for each of these groups. The content of the list box depends on the selected sheet:

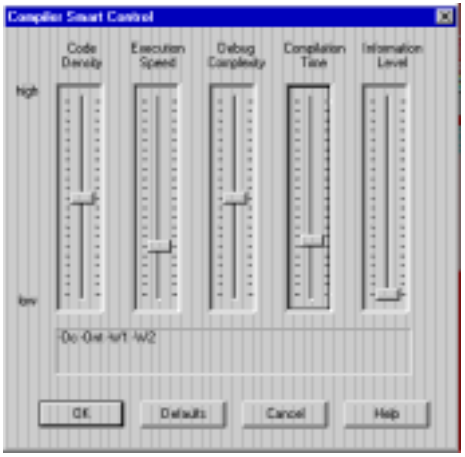
Group	**Description**
Optimization	lists optimization options.
Output	lists options related to the output files generation (which kind of file should be generated).
Input	lists options related to the input file.
Language	lists options related to the programming language (ANSI C, C++)
Target	lists options related to the target processor.
Host	lists options related to the host.
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 ones.

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

Note: Not all groups may be available for each target.

Note: When options requiring additional parameters are selected, an edit box or an additional sub window may be opened, where the additional parameter can be set. For example for the option 'Write statistic output to file...', available in the Output sheet.

Compiler Smart Control Dialog Box



Note: This dialog box is not supported under Win32s, because Microsoft does not support slider controls under Win32s.

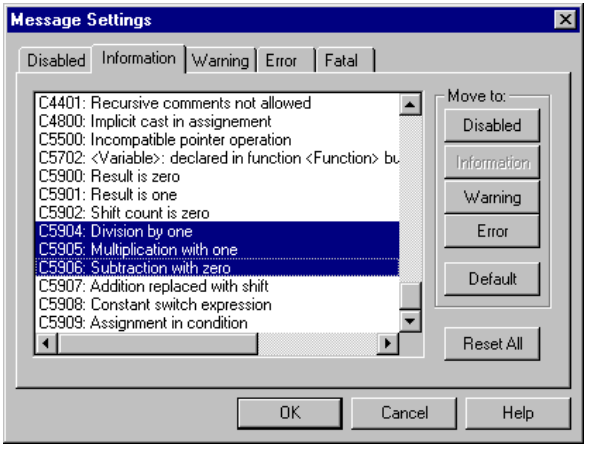
This dialog box allows you to define the optimization level you want to reach during compilation of the input file. Five sliders are available to define the optimization level:

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 high optimization.
Information Level	displays the level of information messages which can be 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 position of the three others is updated according to the modification.

The command line is automatically updated with the options setting an accordance with setting of the different sliders.

Message Settings Dialog Box



This dialog box allows 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. Following buttons are available in the dialog:

Button	Description
Move to: Disabled	The messages selected will be disabled. That is the message will not occur anymore.
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.

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 and the content of the list box depends on the selected sheet:

Message group	Description
Disabled	lists all messages disabled. That means messages displayed in the list box will not be displayed by the Compiler.
Information	lists all information messages. Information messages informs about action taken by the Compiler.
Warning	lists all warning messages. When such a message is generated, compilation of the input file continues and an object file will be generated.
Error	lists all error messages. When such a message is generated, compilation of the input file continues but no object file will be generated.
Fatal	lists all fatal error messages. When such a message is generated, compilation of the input file stops immediately. Fatal messages can not be changed. There are only listed to call context help.

Each message has its own character (‘C’ for Compiler message) 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. In 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 the message '*C1801: Implicit parameter-declaration*' (warning message) as an error message:

- Click the *Warning* sheet, to display the list of all warning messages in the list box.
- Click on the string '*C1801: Implicit parameter-declaration*' in the list box to select the message.
- Click *Error* to define this message as an error message.

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

Note: The 'move to' buttons are only active when only messages are selected which can be moved. 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 with 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, which contains a more detailed description of the error message as well as a small example of code producing it. If several messages are selected, the help of the first is shown. When no message is selected, pressing the key F1 or the help button shows the help for this dialog.

About Box

The about box can be opened with the menu help->about. The about box contains many information. Among others, the current directory and the versions of subparts of the compiler is shown. The main compiler version is displayed separately on top of the dialog.

Click on OK to close this dialog.

Note: During a compilation the subversions of the compiler parts can not be requested. They are only displayed if the compiler currently is not compiling.

Specifying the Input File

There are different ways to specify the input file, which must be compiled. During the compilation, the options will be set according to the configuration performed by the user 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 can be entered in the command line. The specified file will be compiled as soon as the button *Compile* in the tool bar is selected or the enter key is pressed.

Compiling a File which has already been compiled

The command executed previously can be 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 will be compiled as soon as the button *Compile* in the tool bar is selected.

Use the Entry File - Compile...

When the menu entry *File | Compile...* is selected a standard file open file box is displayed, where the user can browse to get the name of the file he wants to compile. The selected file will be compiled as soon as the standard open file box is closed using the *OK* button.

Use Drag and Drop

A file name can be dragged from an external software (for example the File Manager/Explorer) and dropped into the Compiler window. The dropped file will be 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

After compilation there are several ways to check where different errors or warnings have been detected. The format of the error message looks as follows:

```
>> <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 detected.

The user can use his usual editor, to open the source file and correct the errors.

Use a User Defined Editor

The user must first configure the editor he wants to use for message/error feedback in the *Configuration* dialog box. Once a file has been compiled, he only needs to double-click on an error message and the editor selected is automatically activated and points to the line containing the error.

Environment

This part describes the environment variables used by the Compiler. Some of those environment variables are also used by other tools (e.g. Linker), so consult also their respective manual.

Various parameters of the Compiler may be set in an environment using so-called environment variables. The syntax is always the same:

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

Note: Normally no 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
```

These parameters may be defined in several ways:

- Using system environment variables supported by your operating system.
  Putting the definitions in a file called DEFAULT.ENV (.hidefaults for UNIX) in the default directory.

Note: The maximum length of environment variable entries in the DEFAULT.ENV/.hidefaults is 1024 characters.

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

Note: The default directory mentioned above can be set via the system environment variable `DEFAULTDIR`.

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

Note: The environment may also be changed using the -Env Compiler option.

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 / .hidefaults)

Normally, the current directory of a tool started is determined by the operation system or by the program who launches another one (e.g. WinEdit).

For the UNIX operating system, the current directory of an executable started is also the current directory from where the binary file has been started.

For MS Windows based operating systems, the current directory definition is quite complex:

- If the tool is launched using a File Manager/Explorer, the current directory is the location of the executable launched.
- 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 dragging a file on the icon of the executable under Windows 95 or Windows NT 4.0, the desktop is the current directory.
- If the tool is launched by another launching tool with its own current directory specification (e.g. an editor as WinEdit), the current directory is the one specified by the launching tool (e.g. current directory definition in WinEdit).
- For HICROSS + tools, the current directory is where the local project file is in. Changing the current project file does also change 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, the environment variable DEFAULTDIR may be used.

The current directory is displayed among other information with the compiler option “-v” and in the about box.

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

All tools may store some global data into the MCUTOOLS.INI. The tool first search for this 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 located in the MS Windows installation directory (e.g. C:\WINDOS).

Example:

```
C:\WINDOWS\MCUTOOLS.INI
D:\INSTALL\PROG\MCUTOOLS.INI
```

If a tool is started in the D:\INSTALL\PROG\DIRECTOY, the project file in the

same directory than the tool is used (D:\INSTALL\PROG\MCUTOOLS.INI).

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

The following section gives a short description of the entries in the MCUTOOLS.INI file:

[Installation] Section

Entry: Path

Arguments: Last installation path.

Description: Whenever a tool is installed, the installation script stores the installation destination directory into this variable.

Example: Path=c:\install

Entry: Group

Arguments: Last installation program group.

Description: Whenever a tool is installed, the installation script stores the installation program group created into this variable.

Example: Group=ANSI-C Compiler

[Options] Section

Entry: 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

+

Instead of XXX, the actual backend name appears, for example HC11

Entry: SaveOnExit

Arguments: 1/0

Description: 1 if the configuration should be stored when the compiler is closed, 0 if it should not be stored. The compiler does not ask to store a configuration in either cases.

Entry: SaveAppearance

Arguments: 1/0

Description: 1 if the visible topics should be stored when writing a project file, 0 if not. The command line, its history, the windows position and other topics belong to this entry.

Entry: SaveEditor

Arguments: 1/0

Description: 1 if the visible topics should be stored when writing a project file, 0 if not. The editor setting contain all information of the editor configuration dialog.

Entry: SaveOptions

Arguments: 1/0

Description: 1 if the options should be contained when writing a project file, 0 if not. The options do also contain the message settings.

Entry: 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.

Example:

```
SaveOnExit=1
SaveAppearance=1
SaveEditor=1
SaveOptions=1
RecentProject0=C:\myprj\project.ini
RecentProject1=C:\otherprj\project.ini
```

[Editor] Section

+

Entry: 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.

This entry can not be modified with the compiler.

Entry: Editor_Exe

Arguments: The name of the executable file of the global editor

Description: Specifies file name with is called for showing a text file, when the global editor setting is active. In the editor configuration dialog, the global editor selection is only active when this entry is present and not empty.

This entry can not be modified with the compiler.

Entry: Editor_Opts

Arguments: The options to use the global editor

Description: Specifies options, which should be used for the global editor. If this entry is not present or empty, “%f” is used. The command line to launch the editor is build by taking the Editor_Exe content, then appending a space followed by this entry.

This entry can not be modified with the compiler.

Example:

```
[Editor]
editor_name=WinEdit
editor_exe=C:\Winedit\WinEdit.exe
editor_opts=%f
```

Example

The following example shows a typical layout of the MCUTOOLS.INI:

```
[Installation]
```

```
Path=c:\hiware
Group=ANSI-C Compiler

[Editor]
editor_name=WinEdit
editor_exe=C:\Winedit\WinEdit.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
```

Local Configuration File (usually project.ini)

+

The compiler does not change the default.env in any way. Its content is only read. All the configuration properties are instead stored in the configuration file. The same configuration file can and is intended to be used by different applications.

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

The current directory is always the directory, where the configuration is in. If a configuration file in a different directory is loaded, then the current directory also changes. When the current directory changes, also the whole default.env file is

reloaded. Always when a configuration file is loaded or stored, the options in the environment variable COMPOPTIONS is reloaded and added to the project options. This behavior has to be noticed when in different directories different default.env exist which do contain in the COMPOPTIONS incompatible options. When a project is loaded using the first default.env, its COMPOPTIONS are added to the configuration file. If then this configuration is stored in a different directory, where a default.env exists with the incompatible options, the compiler adds options and remarks the inconsistency. Then a message box appears to inform the user that the default.env options were not added. In such a situation the user can either remove the option from the configuration file with the advanced option dialog or he can remove the option from the default.env with the shell or a text editor depending which options should 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 the the current directory

If the option -Prod is used, then the current directory is the directory the project file is in. If the option -Prod is used with a directory, the file project.ini in this directory is loaded.

[Editor] Section

Entry: Editor_Name

Arguments: The name of the local editor

Description: Specifies the name which is displayed for the local editor. This entry has only a description effect. Its content is not used to start the editor.

Saved: This entry can not be modified with the compiler.
This entry has the same format as for the global editor configuration in the mcutools.ini file.

Entry: Editor_Exe

Arguments: The name of the executable file of the local editor

Description: Specifies file name with is called for showing 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: This entry can not be modified with the compiler.
This entry has the same format as for the global editor configuration in the mcutools.ini file.

Entry: Editor_Opts

Arguments: The options to use the local editor

Description: Specifies options, which 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 build by taking the Editor_Exe content, then appending a space followed by this entry.

Saved: This entry can not be modified with the compiler.
This entry has the same format as for the global editor configuration in the mcutools.ini file.

Example:
```
[Editor]
editor_name=WinEdit
editor_exe=C:\Winedit\WinEdit.exe
editor_opts=%f
```

[XXX_Compiler] Section

Instead of XXX, the actual backend name appears.

Entry: RecentCommandLineX, X= integer

Arguments: String with a command line histrory entry, e.g. “fibo.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.

Entry: CurrentCommandLine

Arguments: String with the command line, e.g. “fibo.c -w1”

Description: The currently visible command line content.

Saved: Only with Appearance set in the File->Configuration Save Con-

figuration dialog.

Entry:	StatusbarEnabled
Arguments:	1/0
Special:	This entry is only considered at startup. Later load operations do not use it any more.
Description:	Is currently the statusbar enabled. 1 : the statusbar is visible 0 : the statusbar is hidden
Saved:	Only with Appearance set in the File->Configuration Save Configuration dialog.

Entry:	ToolbarEnabled
Arguments:	1/0
Special:	This entry is only considered at startup. Later load operations do not use it any more.
Description:	Is currently the toolbar enabled. 1 : the toolbar is visible 0 : the toolbar is hidden
Saved:	Only with Appearance set in the File->Configuration Save Configuration dialog.

Entry:	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 numbers contain the position and the state of the window (maximized,..) and other flags.
Saved:	Only with Appearance set in the File->Configuration Save Configuration dialog.

Entry:	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

Entry:	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 when saving a configuration file.

Entry:	ShowTipOfDay
Arguments:	0/1
Description:	Should the Tip of the Day dialog be shown at startup. 1 : it should be shown 0 : no, only when opened in the help menu
Saved:	Always when saving a configuration file.

Entry:	Options
Arguments:	-W2
Description:	The currently active option string. Because also the messages are be contained here, this entry can be very long.
Saved:	Only with Options set in the File->Configuration Save Configuration dialog.

Entry: EditorType

Arguments: 0/1/2/3

Description: This entry specifies which editor configuration possibility 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 also Editor Configuration.

Saved: Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Entry: EditorCommandLine

Arguments: command line, for WinEdit: “C:\Winapps\WinEdit.exe %f /#:%l”

Description: Command line content to open a file. For details see also Editor Configuration.

Saved: Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Entry: EditorDDEClientName

Arguments: client commend, e.g. “[open(%f)]”

Description: Name of the client for DDE editor configuration. For details see also Editor Configuration.

Saved: Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Entry: EditorDDETopicName

Arguments: topic name, e.g. “system”

Description: Name of the topic for DDE editor configuration. For details see also Editor Configuration.

Saved: Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Entry: EditorDDEServiceName

Arguments: service name, e.g. “system”

Description: Name of the service for DDE editor configuration. For details see also Editor Configuration.

Saved: Only with Editor Configuration set in the File->Configuration Save Configuration dialog.

Example

The following example shows a typical layout of the configuration file (usually project.ini) :

```
[Editor]
Editor_Name=WinEdit
Editor_Exe=C:\WinEdit\WinEdit.exe %f /#:%l
Editor_Opts=%f

[XXX_Compiler]
StatusbarEnabled=1
ToolbarEnabled=1
WindowPos=0,1,-1,-1,-1,-1,390,107,1103,643
WindowFont=-16,500,0,Courier
TipFilePos=0
ShowTipOfDay=1
Options=-w1
EditorType=3
RecentCommandLine0=fibo.c -w2
RecentCommandLine1=fibo.c
CurrentCommandLine=fibo.c -w2
EditorDDEClientName=[open(%f)]
EditorDDETopicName=system
EditorDDEServiceName=msdev
EditorCommandLine=C:\WinEdit\WinEdit.exe %f /#:%l
```

Paths

Most environment variables contain path lists telling where to look for files. A path list is a list of directory names separated by semicolons following the syntax below:

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

Example:

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

If a directory name is preceded by an asterisk ("*"), the programs recursively search that whole 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/UNIX environment variables (like GENPATH, LIBPATH, etc.) are used.

We strongly recommend working with WinEdit and setting the environment by means of a DEFAULT.ENV (.hidefaults for UNIX) file in your project directory (This 'project directory' can be set in WinEdit's 'Project Configure...' menu command). This way, you can have different projects in different directories, each with its own environment.

Note: When using WinEdit, do not set the system environment variable DEFAULTDIR. If you do so and this variable does not contain the project directory given in WinEdit's project configuration, files might not be put where you expect them to be put!

Line Continuation

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

Example:

```
COMPOPTIONS=\
-W2 \
-Wpd
```

This is the same as

```
COMPOPTIONS=-W2 -Wpd
```

But this feature may be dangerous using it together with paths, e.g.

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

will result in

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

To avoid such problems, we recommend to 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 is devoted to describing each of the environment variables available for the Compiler. The options are listed in alphabetical order and each is divided into several sections.

Topic	Description
Tools	Lists tools which are using this variable
Synonym	Fore some environment variables a synonym also exists. 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 a 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 effects of the variable where possible. The examples shows an entry in the `default.env` for PC or in the `.hidefaults` for UNIX.
See also	Names related sections.

COMPOPTIONS

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 can be used to globally specify certain options that should always be set, so you don't have to specify them at every compilation. NOTE: it is not recommended to use this environment variable if the HI-CROSS+ 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: 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 may be retrieved from the object files using the decoder.
Example:	COPYRIGHT=Copyright by HIWARE AG
See also:	Environment variable USERNAME Environment variable INCLUDETIME

DEFAULTDIR

PC

DEFAULTDIR: Default Current Directory

Tools:	Compiler, Assembler, Linker, Decoder, Debugger, Librarian, Maker
Synonym:	none.
Syntax:	"DEFAULTDIR=" <directory>.
Arguments:	<directory>: Directory to be the default current directory.
Default:	none.
Description:	With this environment variable the default directory for all tools may be specified. All the tools indicated above will take the directory specified as their current directory instead the one defined by the operating system or launching tool (e.g. editor).
	Note: This is an environment variable on system level (global environment variable) It CANNOT be specified in a default environment file (DEFAULT.ENV/.hidefaults).
Example:	DEFAULTDIR=C:\INSTALL\PROJECT
See also:	Section 'The Current Directory'
	Section MCUTOOLS.INI File

ENVIRONMENT

PC

ENVIRONMENT: Environment File Specification

Tools:	Compiler, Linker, Decoder, Debugger, Librarian, Maker
Synonym:	HIENVIRONMENT
Syntax:	"ENVIRONMENT=" <file>.
Arguments:	<file>: file name with path specification, without spaces
Default:	none.
Description:	This variable has to be specified on system level. Normally the Compiler looks in the current directory for a environment file named default.env (.hidefaults on UNIX). 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 system level (global environment variable) It CANNOT be specified in a default environment file (DEFAULT.ENV/.hidefaults).*
Example:	ENVIRONMENT=\HIWARE\prog\global.env
See also:	none.

ERRORFILE

ERRORFILE: Error File Name Specification

Tools: Compiler, Assembler

Synonym: none.

Syntax: "ERRORFILE=" <file name>.

Arguments: <file name>: File name with possible format specifiers.

Description: The environment variable ERRORFILE specifies the name for the error file (used by the Compiler or assembler).

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').

In case of an illegal error file name, a notification box is shown.

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, e.g. if we compile a file \sources\test.c, an error list file \sources\test.err will be generated.

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

e.g. for a source file test.c, an error list file \dir1\test.err will be generated.

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

e.g. for a source file \dir1\dir2\test.c, an error list file \dir1\dir2\errors.txt will be 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 which looks for an error file called EDOUT:

```
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 use, else the editor cannot find the EDOUT file.

See also: none.

GENPATH

GENPATH: #include “File” Path

Tools:	Compiler, Linker, Decoder, Debugger
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 given by GENPATH and finally in the directories given by LIBPATH. *Note: If a directory specification in this environment variables starts with an asterisk (“*”), the whole directory tree is searched recursively depth first, i.e. all subdirectories and their subdirectories and so on are searched, too. Within one level in the tree, search order of the subdirectories is indeterminate (these is not valid for Win32).*
Example:	GENPATH=\sources\include;..\..\headers;\usr\local\lib
See also:	Environment variable LIBPATH

INCLUDETIME

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:	Normally each object file created contains a time stamp indicating the creation time and data as strings. So 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 SQA reasons a binary file compare has to be performed. Even if the informations in two object files are the same, the files do not match exactly because the time stamps are not the same. To avoid such problems this variable may be set 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 may be retrieved from the object files using the decoder.
Example:	INCLUDETIME=OFF
See also:	Environment variable COPYRIGHT Environment variable USERNAME

LIBRARYPATH

LIBRARYPATH: ‘include <File>’ Path

Tools:	Compiler, ELF tools (Burner, ELF Linker, ELF 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 variables starts with an asterisk (“*”), the whole directory tree is searched recursively depth first, i.e. all subdirectories and their subdirectories and so on are searched, too. Within one level in the tree, search order of the subdirectories is indeterminate (these is not valid for Win32).*
Example:	LIBRARYPATH=\sources\include;..\..\headers;\usr\local\lib
See also:	Environment variable GENPATH Environment variable USELIBPATH Section “Input Files”

OBJPATH

OBJPATH: Object File Path

Tools:	Compiler, Linker, Decoder, Debugger
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 looks for an object file (e.g. the Linker), than 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”

TEXTFAMILY

PC

TEXTFAMILY: Text Font Family

Tools:	Compiler (V3.0.x), Assembler (V3.0.x), Linker, Decoder, Librarian, Maker
Synonym:	HITEXTFAMILY.
Syntax:	"TEXTFAMILY=" <FontName>.
Arguments:	<FontName>: Font family name to be used.
Default:	Terminal
Description:	Defines the font family to use. Default font family is “Terminal”.
Example:	TEXTFAMILY=Times
See also:	Environment variable TEXTSIZE Environment variable TEXTSTYLE Environment variable TEXTKIND

TEXTKIND

PC

TEXTKIND: Text Font Character Set

Tools:	Compiler (V3.0.x), Assembler (V3.0.x), Linker, Decoder, Librarian, Maker
Synonym:	HITEXTKIND
Syntax:	"TEXTKIND=" ("OEM" | "ANSI").
Arguments:	"OEM": OEM font character set is used. "ANSI": ANSI font character set is used.
Default:	OEM
Description:	Gives the character set, either OEM or ANSI. Default is OEM.
Example:	TEXTKIND=ANSI
See also:	Environment variable TEXTFAMILY Environment variable TEXTSIZE Environment variable TEXTSTYLE

TEXTPATH

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

TEXTSIZE

PC

TEXTSIZE: Text Font Size

Tools:	Compiler (V3.0.x), Assembler (V3.0.x), Linker, Decoder, Librarian, Decoder, Maker
Synonym:	HITEXTSIZE
Syntax:	"TEXTSIZE=" <number>.
Arguments:	<number>: Font size to be used.
Default:	14
Description:	Defines the size of that font. Default size is 14 point.
Example:	TEXTSIZE=12
See also:	Environment variable TEXTFAMILY Environment variable TEXTSTYLE Environment variable TEXTKIND

TEXTSTYLE

PC

TEXTSTYLE: Text Font Style

Tools:	Compiler (V3.0.x), Assembler (V3.0.x), Linker, Decoder, Librarian, Maker
Synonym:	HITEXTSTYLE
Syntax:	"TEXTSTYLE=" ("NORMAL" | "BOLD").
Arguments:	"NORMAL": Normal font style is used (not bold or italic).
	"BOLD": Bold font style is used.
Default:	NORMAL
Description:	Defines the font style to be used, can be either NORMAL or BOLD. Default is NORMAL.
Example:	TEXTSTYLE=BOLD
See also:	Environment variable TEXTFAMILY
	Environment variable TEXTSIZE
	Environment variable TEXTKIND

TMP

PC

TMP: Temporary directory

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 has to be created, normally 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 an error message “Cannot create temporary file”.

Note: This is an environment variable on system level (global environment variable) It CANNOT be specified in a default environment file (DEFAULT.ENV/.hidefaults).

Example: TMP=C:\TEMP

See also: Section ‘The Current Directory’

USELIBPATH

USELIBPATH: Using LIBPATH Environment Variable

Tools:	Compiler, Linker, Debugger
Synonym:	none.
Syntax:	"USELIBPATH=" ("OFF" | "ON" | "NO" | "YES").
Arguments:	"ON", "YES": The environment variable LIBPATH will be used by the Compiler to look for system header files <*.h>. "NO", "OFF": The environment variable LIBPATH will not be used by the Compiler.
Default:	ON
Description:	This environment variable allows a flexible usage of the LIBPATH environment variable, because LIBPATH may be used by other software (e.g. version management PVCS).
Example:	USELIBPATH=ON
See also:	Environment variable LIBPATH

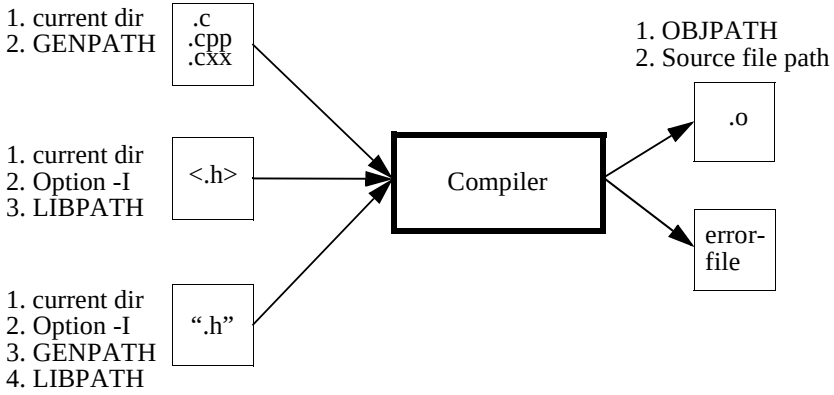
USERNAME

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 may be retrieved from the object files using the decoder.
Example:	USERNAME=HIWARE AG, CH-4058 Basel
See also:	Environment variable COPYRIGHT Environment variable INCLUDETIME

Files

Following figure gives an overview about the file processing 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, we suggest that all your source file names have extension `.c` and all header files extension `.h`. C++ files normally have the extension .cpp or .cxx. Source files will be 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 for 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 given in `LIBPATH` or `LIBRARYPATH`. The current directory is set via WinEdit, the Program Manager or the environment variable `DEFAULTDIR`.

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, then in the directories given in LIBPATH or LIBRARYPATH. The current directory may be set via WinEdit, 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 given in the environment variable OBJPATH. If that variable contains more than one path, the object file is written in the first directory given; if this variable is not set at all, 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 but 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 is displayed, too.

If the Compiler is started from WinEdit (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 EDOUT using the Microsoft format by default. Use WinEdit's *Next Error* or Codewright's *Find Next Error* command to see both 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 EDOUT is generated in the current directory.

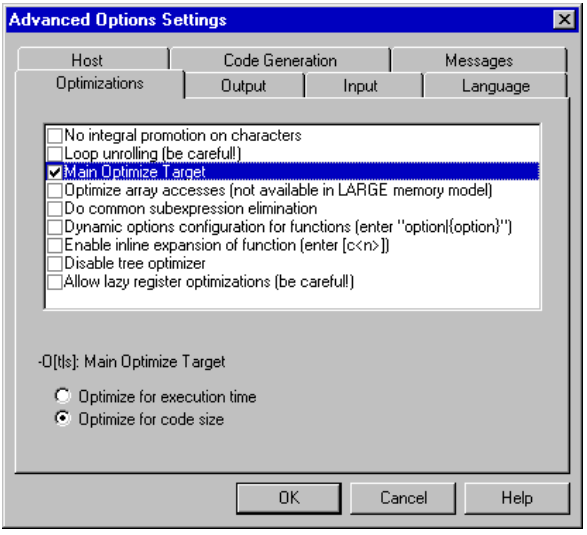
Compiler Options

The Compiler offers a number of Compiler options that you can use to control the Compiler's operation. Options are composed of a minus/dash ('-') followed by one or more letters or digits. Anything not starting with a dash/minus is supposed to be the name of a source file to be compiled. Compiler options may be specified on the command line or in the COMPOPTIONS variable. Typically, each Compiler option is specified only once per compilation.

Command line options are not case sensitive, e.g. "`–Li`" is the same as "`–li`". To facilitate the specification of several options belonging to the same group, e.g. "`–Cc`" and "`–Cu`", the Compiler allows coalescing options in the same group, i.e. one might also write "`–Ccu`" or "`–Cuc`" instead of "`–Cc –Cu`".

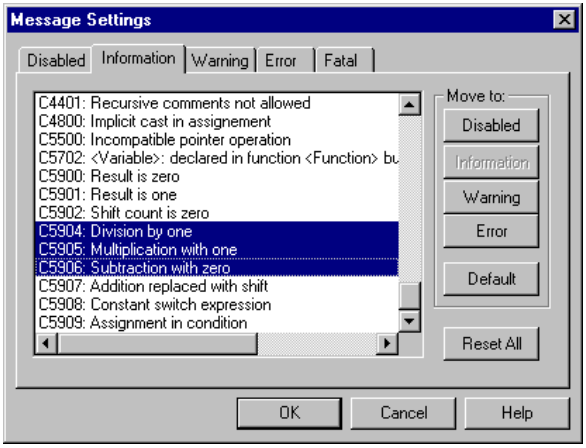
Note: It is not possible to coalesce options in different groups, e.g. "`–Cc –Li`" cannot be abbreviated by the terms "`–Cci`" or "`–CcLi`"!

Another way to set the Compiler options is to use the Graphical User Interface:



NOTE: Don't use the environment variable COMPOPTIONS if the graphical user interface is used. The Compiler stores the options in the project.ini, NOT in the default.env.

To move messages (options -Wmsg), the dialog below may be used too:



Option Recommendation

Depending on the compiled sources, each Compiler optimization may have his advantages or disadvantages. It is not easy to provide a general recommendation of option settings to get the best result, but some of them normally make always sense:

-Cc (allocate constant objects into ROM), but do not forget 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 may be the source of problems in your code (e.g. they make the code hard to debug), so just switch them off (the options usually have the prefix -On), candidates for such optimizations are peephole optimizations.

Some optimizations may produce for some functions more code, for other ones less code (e.g. inlining (-Oi, only available with HI-CROSS+), loop unrolling (-Cu), CSE (-Oc)). Try those options to get the best result for each.

For best results for each function, try to compile each module with the -OdocF. A usual example for the this option is -OdocF="-Or|-Oc".

Compiler Option Details

Option Groups

Compiler options are grouped by:
HOST, LANGUAGE, OPTIMIZATION, OUTPUT, INPUT, CODE, TARGET, MESSAGE and VARIOUS

Group	Description
HOST	Lists options related to the host.
LANGUAGE	Lists options related to the programming language (ANSI C, C++, ...)
OPTIMIZATION	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	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.

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

Note: Not all command line options are accessible trough the property sheets, because they have a special graphical setting (e.g. the option to set the type sizes).

Option Scopes

Each option has also a scope:

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 can be 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 options for the message management.

The options available are arranged into different groups, and 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 is devoted to describing each of the Compiler options available for the Compiler. The options are listed in alphabetical order and each is divided into several sections.

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

Topic	Description
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 PC or in the `.hidefaults` for UNIX.
See also	Names related options.

-!

-!: 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:	`COMPOPTIONS=-!`

This truncates the following include directive

```
#include “mylongfilename.h”
```

to

```
#include “mylongfi.h”
```

See also:	none.

-Ansi

-Ansi: Strict ANSI

Group:	LANGUAGE
Scope:	Function
Syntax:	"-Ansi".
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. far and near) are not accepted by the Compiler, and the Compiler will generate an error. The Compiler defines __STDC__ if this option is set.
Example:	`COMPOPTIONS=-ANSI`
See also:	none.

-BfaB

HC08

-BfaB: Bit Field Byte Allocation

Group: CODE GENERATION

Scope: Application

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)

Default: -BfaBLS

Defines: __BITFIELD_MSBYTE_FIRST__
__BITFIELD_MSBIT_FIRST__
__BITFIELD_LSBIT_FIRST__
__BITFIELD_LSBYTE_FIRST__

Pragmas: none.

Description: Normally, bits in byte bitfields are allocated from 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, this option may be used.

Example: `COMPOPTIONS=-BfaBMS`

See also: Bit Field Allocation Defines

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

C++

-C++: Enable C++ Support

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 kinds 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 option -C++f were set.
Example:	`COMPOTIONS=-C++f`
See also:	C++ Front End Option -Cn

-Ca

HC08

-Ca: 7bit Pointer Difference

Group:	OPTIMIZATION
Scope:	Function
Syntax:	"-Ca".
Arguments:	none.
Default:	none.
Defines:	none.
Pragmas:	none.
Description:	Optimization of pointer additions: The Compiler may assume that all offsets in pointer additions lie in the range [0..127].
	*Note: In ANSI-C array operations are in fact pointer operations. e.g. arr[3] is the same as *(arr+3). If this option is set the arrays must not exceed 127 bytes.*
Example:	`COMPOPTIONS=-Ca`
See also:	none.

-Cc

-Cc: Allocate Constant Objects into ROM

Group: OUTPUT

Scope: Compilation Unit

Syntax: "-Cc".

Arguments: none.

Default: none.

Defines: none.

Pragmas: #pragma INTO_ROM

Description: Variables declared as `const` are treated just like any other variable, unless the command line option "–`Cc`" has been given: then, `const` objects are put into segment `ROM_VAR`, which then may be assigned a ROM section in the Linker parameter file (please see document *Linker*).

For objects allocated into a read-only section, the Linker prepares no initialization, thus the startup code has not to copy down the constant data.

Of course you may also put variables into segment `ROM_VAR` by using the segment pragma (please see document *Linker*).

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 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 (please see chapter *Linker* for more information).

Note: This option is useful only for HIWARE object file format, in the ELF/DWARF object file format constants are allocated into the ELF section `".rodata"`

Note: The Compiler uses the default addressing mode for the

constants specified by the memory model.

Example: `COMPOPTIONS=-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

-Ci

+

-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 can be replaced with the following keywords:

Bigraph		Trigraph		Additional Keyword	
`<%`	`}`	`??=`	`#`	`and`	`&&`
`%>`	`}`	`??/`	`\`	`and_eq`	`&=`
`<:`	`[`	`??'`	`^`	`bitand`	`&`
`:>`	`]`	`??(`	`[`	`bitor`	`|`
`%:`	`#`	`??)`	`]`	`compl`	`~`
`%:%:`	`##`	`??!`	`|`	`not`	`!`
		`??<`	`{`	`or`	`||`
		`??>`	`}`	`or_eq`	`|=`
		`??-`	`~`	`xor`	`^`
				`xor_eq`	`^=`
				`not_eq`	`!=`

Note: The additional keywords are not allowed as identifier if

this option is enabled.

Example: COMPOPTIONS=-Ci

This example shows the usage of trigraphs, bigraphs and the additional keywords with the corresponding ‘normal’ C-source.

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

See also: none.

-Cn

C++

-Cn: Disable compactC++ features

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 much memory.

"Tpl" Templates are not allowed.
Avoid having many generated functions doing all almost the same.

"Ptm" Pointer to member not allowed.
Avoid having pointer to member objects consuming much memory.

"Mih" Multiple inheritance is not allowed.
Avoid having complex class hierarchies.
Because virtual base classes make only sense with multiple inheritance, they are not allowed, either.

"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 can be useful for compiling C sources with the C++ Compiler, if you don't want C structures to get 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: `COMPOPTIONS=-C++c -Cn=Ctr`

See also: Option -C++c
C++ Front End.

-Cni

-Cni: No Integral Promotion

Group: OPIMIZATION

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 none ANSI-C compliant behavior.
In ANSI-C operations with data types smaller than int has to be promoted to int (integral promotion). With this rule, adding two unsigned character variables will result in zero-extension of each character operand and then adding them as int operands. But if the result has to be stored back into a character, this integral promotion is not necessary, and with this option set this promotion is avoided where possible.
Because operations may be performed on character base instead on integer base with this option set, code size will be decreased.

The –Cni option enhances code density of character operations by omitting integral promotion.

Consider the following:

In 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 is not ANSI standard conforming, and thus code compiled with this option is not portable.

The Compiler recognizes, with the -Cni option enabled that a char multiplication results an integer. (The ANSI behavior would be to evaluate the expression using an int multiplication.) Consider the following example:

Example (with pseudo code):

```
int a; char b, c;

a = b*c;     MOV    R1L,b
             MOV    R2L,c
             MULU.b R1L,R2L
             MOV.w   a,R1
```

'Old style declarations' of functions using the char parameter are required to be extended to int by the ANSI standard. The -Cni option disables this extension and thus saves 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, no matter 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 arithmetic conversions of ANSI-C are applied.

If constants are in the character range they are treated as characters. Remember that the type char is unsigned and thus this 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 have to be casted.

Example:

```
signed char a, b;

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

Note: This option is ignored with the –`ANSI` *Compiler switch active.*

Example: `COMPOPTIONS=-Cni`

See also: none.

-Cq

-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 itself 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, too.
Example:	`COMPOPTIONS=-Cq`
See also:	none.

-CswMaxLF

+

-CswMaxLB: 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: 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 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 enough, a branch tree is generated (tree of if-elseif-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 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 much easier.

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, just set the minimum load factor needed for a table to 100 and the maximum load factor to 100 too: -CswMinLF100 -CswMaxLF100.

Example: `COMPOPTIONS=-CswMaxLF50`

See also: Option -CswMinLB
Option -CswMinSLB
Option -CswMinLF

-CswMinLB

+

-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 back-end dependent). If there are not enough labels for a table, a branch tree is generated (tree of if-elseif-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 a table may code execution speed, but it probably increase 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 makes usually since also to disable search tables with option -CswMinSLB.

Example: `COMPOPTIONS=-CswMinLB5`

See also: Option -CswMinLF
Option -CswMinSLB
Option -CswMaxLF

-CswMinSLB

+

-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:	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 search tables.

Switch tables may be implemented in different ways. When almost all case entries in some range are given, then a table only containing branch targets can be used. Using such a dense table can be time efficiently encoded by just accessing the right entry. When large wholes exists in some areas, still a table form can be used. But now the case entry and its corresponding branch target are encoded in the table. This is called a search table. Now to access such a search table a complex runtime routine must be used. This routine checks all entries until it finds the current one. So search tables are quite slow in execution.

Using a search table does usually improves code density, but the execution time significantly increases. Every time some entry in a search table must be found, all previous entries have first to be checked. For a dense table, just the right offset can be computed and accessed. Additionally note that all backends do 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, just set the minimum number of labels needed for a table to a high value (e.g. 9999): -CswMinSLB9999.

Example: `COMPOPTIONS=-CswMinLB5`

See also: Option -CswMinLB
Option -CswMinLF
Option -CswMaxLF

-CswMinLF

+

-CswMinLB: 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: 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 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 enough, a branch tree is generated (tree of if-elseif-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 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 much easier.

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 maximum load factor is set to 100% and setting the minimum load factor for the above case to 90% still will generate a table. But setting the minimum load factor to 95% will produce a branch tree.

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

Example: `COMPOPTIONS=-CswMinLF50`

See also: Option -CswMinLB
Option -CswMinSLB
Option -CswMaxLF

-Cu

-Cu: Loop Unrolling

Group: OPTIMIZATION

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>. E.g. -Cu=i20 unrolls loops with a maximum of 20 iterations.

Example: `COMPOPTIONS=-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 transforms also 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.

-Cv

HC08

-Cv: Local Variables of Size 2 in _OVERLAP

Group:	OPTIMIZATION
Scope:	Function
Syntax:	"-Cv".
Arguments:	none.
Default:	none.
Defines:	none.
Pragmas:	none.
Description:	Allocates local variables of size 2 into the _OVERLAP segment. See Back End for details.
Example:	`COMPOPTIONS=-Cv`
See also:	none.

-Cx

-Cx: No Code Generation

Group:	OUTPUT
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, but the Compiler does perform a syntactical check of the source code. This allows a quick test if the Compiler accepts the sources without errors.
Example:	`COMPOPTIONS=-Cx`
See also:	none.

-D

-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:	`COMPOPTIONS=-DDEBUG=0`
	This is the same as writing
	`#define DEBUG 0`
	in the source file.
	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

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

Group: LANGUAGE

Scope: Function

Syntax: "-Ec".

Arguments: none.

Default: none.

Description: If this not 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 trough a pointer, so this option is useful if some older sources 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: `COMPOPTIONS=-Ec`

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

See also: none.

-Env

-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.
Defines:	none.
Pragmas:	none.

Example:	`COMPOPTIONS=-EnvOBJPATH=\sources\obj`
	This is the same as
	`OBJPATH=\sources\obj`
	in the default.env

See also:	Section “Environment”

-F (-Fh, -F1, -F2, -F7)

+

-F: Object File Format

Group: OUTPUT

Scope: Application

Syntax: "-F" ("h" | "1" | "2" | "7").

Arguments: "h": HIWARE object file format, this is the default
"1": ELF/DWARF 1.1 object file format
"2": ELF/DWARF 2.0 object file format
"7": strict HIWARE V2.7 object file format

Note: Not all object file formats may be available for a target, the HIWARE object file format is used by default.

Default: -Fh

Defines: __HIWARE_OBJECT_FILE_FORMAT__
__ELF_OBJECT_FILE_FORMAT__

Pragmas: none.

Description: The Compiler writes the code and debugging info after compilation into a object file.
With the option -Fh (which is also the default) or -F7 set, the Compiler uses an object file format which is proprietary of HIWARE.
The HI-CROSS 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 zero bytes in strings allowed (zero byte marks the end of the string)
The strict HI-CROSS 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)
With the options -F1 or -F2 set, the Compiler produces an ELF/DWARF object file. This object file formats may also be supported by other Compiler vendors.

Note: To use the ELF/DWARF object file format, you have to use also a special Linker, Decoder and Debugger.

Example: `COMPOPTIONS=-Fh`

See also: none.

-Fd

HC08

-Fd: Doubles are IEEE32

Group:	CODE GENERATION
Scope:	Application
Syntax:	"-Fd".
Arguments:	none
Default:	none.
Defines:	see Option -T.
Pragmas:	see Option -T.
Description:	This option allows changing the float/double format. By default, float is IEEE32 and doubles are IEEE64. With this option set, all doubles are IEEE32 format instead IEEE64. Floating point formats may be also changed with the -T option.
Example:	`COMPOPTIONS=-Fd`
See also:	Option -T

-H

-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, OPTIMIZATION, 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

-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:	This option allows setting of include paths in addition to the LIBPATH, LIBRARYPATH and GENPATH environment variables. Paths specified with this option have precedence over paths specified in GENPATH, LIBPATH and LIBRARYPATH.
Example:	`COMPOPTIONS=-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 continuous as normal 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”
	Environment variable LIBPATH
	Environment variable LIBRARYPATH
	Environment variable GENPATH.

-La

-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 the following modifiers :

Modifier	Replaced by
%p	path
%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 single %

Default: No file created.

Defines: none.

Pragmas: none.

Description: The -La option causes the Compiler to generate a assembler include file when a pragma CREATE_ASM_LISTING occurs. The name of the file created is specified by this option. If no name is specified, a default of “%f.inc” is taken and the environment variable TEXTPATH is used if the result file name contains no path information.
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: `COMPOPTIONS=-La=asm.inc`

See also: pragma CREATE_ASM_LISTING

Generating an Assembler Include File

-Li

-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 extension `.INC`. *Note: No code will be generated if this option has been set.*
Example:	`COMPOPTIONS=-Li`
See also:	Option -Lm

-Lic

-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:	`COMPOPTIONS=-Lic`
See also:	Option -LicA

-LicA

-Lic: 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 of every tool or dll in the directory were the executable is (e.g. if tool or feature is a demo version or a full version). Because the option has to analyse every single file in the diretory, this could be a lenghty process
Example:	`COMPOPTIONS=-LicA`
See also:	Option -Lic

-Ldf

-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 which contains a list of the Compiler defined `#define`. The default file name is ‘predef.h’, but may be changed by the user (e.g. -Ldf=”myfile.h”. The file is generated in the directory specified by the environment variable TEXTPATH or if this variable does not exist in the current directory. The defines written to this file depend als 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 the sources.

Note: This option only has an effect if a file is compiled. Just using this option without to compile a file is not possible.

Example: `COMPOPTIONS=-Ldf`

This generates the file ‘predef.h’ with 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

-Ll

-Ll: Statistics about each Function

Group: OUTPUT

Scope: Compilation Unit

Syntax: "-Ll" ["=" <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 `-Ll` 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, without return address!) 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 environment variable TEXTPATH is set, the file is stored into the path specified by the environment variable, else into the current directory.

Example: `COMPOPTIONS=-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

-Lm: Include Files in Make File List

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 make format. The `-Lm` option is very useful in aiding in creating make files. The output from several source files may be copied and grouped into one make file. The generated list is in 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 environment variable TEXTPATH is set, the file is stored into the path specified by the environment variable, else into the current directory.

Note: No code will be generated if this option has been set.

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

-Lo

-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 the environment variable TEXTPATH is set, the file is stored into the path specified by the environment variable, else into the current directory. *Note: No code will be generated if this option has been set.*
Example:	`COMPOPTIONS=-Lo`
See also:	Option -Li Option -Lm

-Lp

-Lp: Preprocessor Output

Group: OUTPUT

Scope: Compilation Unit

Syntax: "-Lp" ["=" <filename>].

Arguments: <filename> : The name of the file to be generated.
It may contain the following modifiers :

Modifier	Replaced by
%p	path
%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 single %

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 in the option, this text file shares the same name as the source file but with extension `.PRE` (%n.pre). The environment variable TEXTPATH 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, and only source code is listed with line numbers.

Example: `COMPOPTIONS=-Lp`

See also: none.

-M (-Ms, -Mb)

HC08

-M: Memory Model

Group:	CODE GENERATION
Scope:	Application
Syntax:	"-M" ("s" | "b").
Arguments:	"s": small memory model "b": banked memory model.
Default:	-Ms
Defines:	__SMALL__ __BANKED__
Pragmas:	none.
Description:	See Back End for details
Example:	`COMPOPTIONS=-Ms`
See also:	none.

-N

PC

-N: Display Notify Box

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 the user 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: `COMPOPTIONS=-N`

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

See also: none.

-NoBeep

-NoBeep: No Beep in Case of an Error

Group:	MESSAGE
Scope:	Function
Syntax:	"-NoBeep".
Arguments:	none.
Default:	none.
Defines:	none.
Pragmas:	none.
Description:	Normally there is a ‘beep’ notification at the end of processing if there was an error. To have a silent error behaviour, this ‘beep’ may be switched of using this option.
Example:	`COMPOPTIONS=-NoBeep`
See also:	none.

-ObjN

-ObjN: Object file name specification

Group: OUTPUT

Scope: Compilation Unit

Syntax: "-ObjN="<file>.

Arguments: <file>: Object file name.

Default: -ObjN=%n.o

Defines: none.

Pragmas: none.

Description: Normally, the object file has the same name than the processed source file, but with extension ".o". This option allows a flexible way to define the object file name. The modifier "%n" can be used, it is replaced with the source file name. If <file> in the option contains a path (absolute or relative), the environment variable OBJPATH is ignored.

Example: `COMPOPTIONS=-ObjN=a.out`

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

```
fibo.c -ObjN=%n.obj
```

The resulting object file will be "fibo.obj".

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

The object file will be named relative to the current directory to "..\objects_myfile.obj. Note that the environment variable OBJPATH is ignored, because the <file> contains a path.

See also: Environment variable OBJPATH

-Oc

+

-Oc: Common Subexpression Elimination (CSE)

Group: OPTIMIZATION

Scope: Function

Syntax: "-Oc".

Arguments: none.

Default: none.

Defines: none.

Pragmas: none.

Description: Do common subexpression elimination (CSE). The code for common subexpressions and assignments is generated only once and the result of it is reused. Depending on available registers, a common subexpression may produce more code because of many spills.

Note: The CSE switched on is not always safe. Changes of variables by aliases may generate incorrect optimizations.

Example: `COMPOPTIONS=-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.

-OdocF

-OdocF: Dynamic Option Configuration for Functions

Group: OPTIMIZATION

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, a developer has to set a specific set of Compiler switches for each compilation unit (file to be compiled). The problem is that for some files a specific set of options may decrease the code size, but for another file the same set of Compiler options may produce more code depending on the sources.

Unfortunately, some optimizations may reduce the code size for some functions, but may increase the code size for other functions in the same compilation unit. But normally it is impossible to vary options over different functions or to find out the best combination of options. This option solves this problem. This option allows the Compiler to choose from a set of options to reach the smallest code size for every function. Without this feature, the programmer has to set some Compiler switches which are fixed over the hole compilation unit. With this feature the Compiler is free to find out 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"
```

With this feature enabled, the Compiler tries to find out of the specified set the options the best option combination. The Compiler evaluates all possible combinations out of the sets specified, e.g. for the following option

```
-W2 -OdocF="-Or|-Cniu|-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 hole compilation unit (like inlining or macro definition) should be specified in this option, because 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: `COMPOPTIONS=-Odocf=”-Or|-Cni”`

See also: none.

-Oi

+

-Oi: Inline Expansion of Functions

Group: OPTIMIZATION

Scope: Compilation unit

Syntax: "-Oi" ["=" <code Size>].

Arguments: <code Size>: Limit for inlining in code size.

Default: none.

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.

Example: `COMPOPTIONS=-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>] means, all functions with a size smaller than <n> are inlined. e.g. compiling with the option `-oi=c100` enables inline expansion for all functions with a size smaller than 100 bytes.

Restrictions: Following functions are not inlined:

- functions with function calls inside (“non-leaf”-functions)
- functions with default arguments
- functions with labels inside
- functions with inline assembly statements

- functions using local static objects

See also: none.

-OnP

HC08

-OnP: Disable peephole Optimization

Group: OPTIMIZATION

Scope: Function

Syntax: "-OnP" ["=" <option Char> {<option Char>}].

Arguments: <option Char> is one of the following:

"a": Disable JSR; RTS to JMP optimization.

Default: none.

Defines: none.

Pragmas: none.

Description: If -OnP is specified, the hole peephole optimizer is disabled. To disable only a single peephole optimization, the optional syntax -OnP=<char> may be used, e.g.

Suboption -OnP=a (Disable JSR; RTS to JMP optimization) :

The compiler replaces a JSR- RTS sequence with a single JMP instruction.
This saves 1 byte code- and 2 bytes stack space.
To avoid this optimization, use the option -onp=a.

Code Example:

With -onp=a:

```
...
JSR  Subroutine
RTS
```

Without -onp=a:

```
...
JMP  Subroutine
```

Example: `COMPOPTIONS=-OnP=a`

Disables the peephole optimization ‘a’

See also: none.

-Ont

+

-Ont: Disable Tree Optimizer

Group: OPTIMIZATION

Scope: Function

Syntax: "-Ont".

Arguments: none.

Default: none.

Defines: none.

Pragmas: none.

Description: The Compiler contains a special optimizer which optimizes the internal tree data structure. This tree data structure is used to hold the semantic of the program and represent more or less the statements and expressions parsed.

This option can be used to disable this tree optimizer, this may be useful for debugging reasons and to force the Compiler to produce ‘straight forward’ code.

If this option is set, the Compiler will not perform the following optimizations:

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;
```

Reduction of long compares into int compares if int compares are better:

```
if (uL == 0)
```

will be optimized into

```
if ((int)(uL>>16) == 0 && (int)uL == 0)
```

Example: `COMPOPTIONS=-Ont`

See also: none.

-Or

HC08

-Or: Lazy Register Optimization

Group:	OPTIMIZATION
Scope:	Function
Syntax:	"-Or".
Arguments:	none.
Default:	none.
Defines:	none.
Pragmas:	none.
Description:	Allow lazy register tracing resulting in denser code. See Back End for details.
Example:	`COMPOPTIONS=-Or`
See also:	none.

-O (-Os, -Ot)

-Os/-Ot: Main Optimization Target

Group: OPTIMIZATION

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 will generally optimize on code size; it often has to decide between a runtime routine and expanded code. The programmer can decide, whether to choose 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 will “trade” 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: `COMPOPTIONS=-Os`

See also: none.

-Pe

-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 will be handled in an include directive just the same as like in a printf instruction:

```
#define STRING “c:\myfile.h”
#include STRING
```

will produce an error:

```
>> Illegal escape sequence
```

and used in:

```
printf(STRING);
```

will produce a carriage return with line feed:

```
c:
myfile
```

If the option -Pe is used, escape sequences are ignored in strings that contains a DOS drive letter ('a..'z', 'A'..'Z') followed by a colon ':' and a backslash '\'

With the -Pe option the Compiler handles string in include directive differently from other strings, that means that escape sequences in include directive strings are not evaluated, e.g.

```
#include "c:\names.h"
```

results in exact the same include file name as in the source

(“`c:\names.h`”). But if the file name appears in a macro, the Compiler has no chance to distinguish between file name usage and normal string usage with escape sequence, because the macro STRING has to be the same for the include and the printf call:

```
#define STRING "c:\n.h"
#include STRING /* means: "c:\n.h" */

void main(void) {
  printf(STRING);/* means: "c:", new line and ".h" */
}
```

To use macros for include files this option may be used. The 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 will include the file `'c:\n.h'` and calls printf with `"c:\n.h"`).

Example: `COMPOPTIONS=-Pe`

See also: none.

-Pio

-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 a #include directive, he 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 :

```
/* Header file myfile.h */
#ifndef _MY_FILE_H_
#define _MY_FILE_H_

/* .... content .... */

#endif /* _MY_FILE_H_ */
```

These #ifndef and #defines have the effect that even when the header file is included several times, its content is read only the first time. This solves many problems in C as it is not allowed to define structures, enums or typedefs twice.

When all header files are protected this way, then this option can safely accelerate the compilation.

This option however must not be used, when a header file must be included twice, for example because it contains macros which are set differently at the different inclusion times.
In such a case the #pragma ONCE can be used to accelerate the inclusion of save header files, which do not contain such parts.

Example: `COMPOPTIONS=-Pio`

See also: #pragma ONCE

-Prod

+

-Prod: specify project file at startup

Group:	none. (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 compiler. 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 compiler 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”)
See also:	Section Configuration File.

-Qvtp

C++

-Qvtp: Qualifier for virtual table pointers

Group: CODE GENERATION

Scope: Application

Syntax: "-Qvtp" ("none" | "far" | "near" | "rom" | "uni" | "paged").

Arguments: none.

Default: -Qvptnone

Defines: none.

Pragmas: none.

Description: Using virtual function in C++ needs an additional pointer to virtual function tables. This pointer is not accessible for the programmer, this pointer is generated by the compiler in every class object with virtual function tables 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: `COMPOPTIONS=-QvtpFar`

This sets the qualifier for virtual table pointers to far, thus the virtual tables may be placed into a far segment (if the Back End/CPU does support far segments).

See also: C++ Front End

-T

+

-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__

__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__
```

Pragmas: none.

Description: This option allows configurable type settings. The syntax of the option is: -T{<type> <format>}

For <type> one of the following keys may be specified:

Type	Key
char	'c'
short	's'
int	'i'
long	'L'
long long	'LL'
float	'f'
double	'd'
long double	'Ld'
long long double	'LLd'

Type	Key
enum	'e'
virtual table delta size	'vtd'

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 before the key for the char key:

Sign prefix	Key
signed	's'
unsigned	'u'

for <format> one of the following keys may be specified:

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. E.g. it is illegal if no type at all is set to a size of one. See Back End for default settings.

Note: Enumeration types have the type ‘signed int’ by default for ANSI-C compliance. Be careful if you change this

ANSI default.

Example: -Tsc sets ´char´ to ´signed char´
and -Tuc sets ´char´ to ´unsigned char´

Example:

```
-Tsc1s2i2L4LL4f2d4Ld4LLd4e2
```

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)
```

e.g. it is illegal to set `char` to 16 bit and `int` to 8 bit.

Be careful if you change the sizes. Type sizes has to be consistent over the hole application, e.g. using different type sizes may result in a mess. Note that the libraries delivered with the Compiler are compiled with the standard type settings.
Be also aware for under/overflows if you change the type sizes, e.g. assigning a value too large to an object which is smaller now:

```
int i; /* -Ti1 int has been set to 8 bit! */
i = 0x1234;   /* i will set to 0x34! */
```

Examples:

Setting the size of char to 16 bit:
`-Tc2`

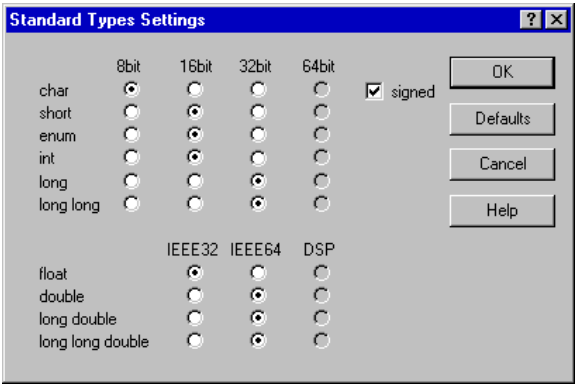
Setting the size of char to 16 bit and plain char is signed:
`-Tsc2`

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 option -Tvtd allows 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: none.

-V

-V: Prints the Compiler Version

Group: VARIOUS

Scope: None

Syntax: "-V".

Arguments: none.

Default: none.

Defines: none.

Pragmas: none.

Description: Prints the Compiler version, internal subversion numbers of parts the Compiler consists of and the current directory.

Note: This option is useful to 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.

-Wmsg8x3

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) are expecting the file name in the Microsoft message format in a strict 8.3 format, that means the file name can have at most 8 characters with not more than a 3 characters extension. Using Win95 or WinNT longer file names are possible. With this option the file name in the Microsoft message is truncated to the 8.3 format.

Example:

```
x:\mysourcefile.c(3): INFORMATION C2901: Unrolling
loop
```

With the option -Wmsg8x3 set, the above message will be

```
x:\mysource.c(3): INFORMATION C2901: Unrolling loop
```

See also: Option -WmsgFi
Option -WmsgFb

-WmsgFb (-WmsgFbi, -WmsgFbm)

+

-WmsgFb: 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: The Compiler can be started 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 WinEdit), the Compiler compiles the files in a batch mode, that is 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 to the screen. This file only contains the compiler messages (see examples below).

By default, the Compiler uses a Microsoft message format to write the Compiler messages (errors, warnings, information messages) if the compiler is in batch mode.

With this option, the default format may be changed 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++);
}
```

By default, the Compiler may produce 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:

```
COMPOPTIONS=-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)

+

-WmsgFi: 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: If the Compiler is started without additional arguments (e.g. files to be compiled together with Compiler options), the Compiler is in the interactive mode (that is, a window is visible).

By default, the Compiler uses the verbose error file format to write the Compiler messages (errors, warnings, information messages).

With this option, the default format may be changed 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++);
}
```

By default, the Compiler may produce 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:\CHC11\WORK\HC11\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

Code Size: 11
```

Setting the format to Microsoft, less information is displayed:

`COMPOPTIONS=-WmsgFim`

```
Top: X:\C.C
Object File: X:\C.O
X:\C.C(3): INFORMATION C2901: Unrolling loop
X:\C.C(2): INFORMATION C5702: j: declared in function
foo but not referenced
Code Size: 11
```

See also: Environment variable ERRORFILE

Option -WmsgFb

-WmsgFob

-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: With this option it is possible modify the default message format in batch mode. Following formats are supported (supposed that the source file is x:\hiware\mysourcefile.cpph)

```
Format Description        Example
-----------------------------------
%s     Source Extract
%p     Path                x:\hiware\
%f     Path and name       x:\hiware\source
%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     Upercase kind       ERROR
%k     Lowercase kind      error
%d     Number              C1815
%m     Message             text
%%     Percent             %
\n     New line
```

Example: `COMPOPTIONS=-WmsgFob”%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 -WmsgFoi

-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: With this option it is possible modify the default message format in interactive mode. Following formats are supported (supposed that the source file is x:\hiware\mysourcefile.cpph):

```
Format Description        Example
-----------------------------------
%s     Source Extract
%p     Path                x:\hiware\
%f     Path and name       x:\hiware\source
%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     Upercase kind       ERROR
%k     Lowercase kind      error
%d     Number              C1815
%m     Message             text
%%     Percent             %
\n     New line
```

Example: `COMPOPTIONS=-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

-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 this message format string is used. Following formats are supported:

```
Format Description        Example
-----------------------------------
%K     Upercase kind      ERROR
%k     Lowercase kind     error
%d     Number             C1815
%m     Message            text
%%     Percent            %
\n     New line
```

Example: `COMPOPTIONS=-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

-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 this message format string is used. Following formats are supported (supposed that the source file is x:\hiware\mysourcefile.cpph)

```
Format Description        Example
-----------------------------------
%p     Path                x:\hiware\
%f     Path and name       x:\hiware\source
%n     File name           mysourcefile
%e     Extension           .cpph
%N     File (8 chars)      mysource
%E     Extension (3 chars) .cpp
%K     Upercase kind       ERROR
%k     Lowercase kind      error
%d     Number              C1815
%m     Message             text
%%     Percent             %
\n     New line
```

Example: `COMPOPTIONS=-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

+

-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:	With this option the amount of error messages can be set until the Compiler stops compilation. Note that subsequent error messages which depends on a previous one may be confusing.
Example:	`COMPOPTIONS=-WmsgNe2` The Compiler stops compilation after two error messages.
See also:	Option -WmsgNi Option -WmsgNw

-WmsgNi

+

-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:	With this option the amount of information messages can be set.
Example:	`COMPOPTIONS=-WmsgNi10`
	Only ten information messages are logged.
See also:	Option -WmsgNe Option -WmsgNw

-WmsgNw

+

-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:	With this option the amount of warning messages can be set.
Example:	`COMPOPTIONS=-WmsgNw15` Only 15 warning messages are logged.
See also:	Option -WmsgNe Option -WmsgNi

-WmsgSd

+

-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:	With this option a message can be disabled, so it does not appear in the error output. This option cannot be used in a pragma OPTION. Use the pragma MESSAGE instead.
Example:	`COMPOPTIONS=-WmsgSd1801` disables the message for implicit parameter declaration.
See also:	Option -WmsgSi Option -WmsgSw Option -WmsgSe Pragma MESSAGE

-WmsgSe

+

-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:	Allows changing a message to an error message. This option cannot be used in a pragma OPTION. Use the pragma MESSAGE instead.
Example:	`COMPOTIONS=-WmsgSe1853`
See also:	Option -WmsgSd Option -WmsgSi Option -WmsgSw Pragma MESSAGE

-WmsgSi

+

-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:	With this option a message can be set to an information message. This option cannot be used in a pragma OPTION. Use the pragma MESSAGE instead.
Example:	`COMPOPTIONS=-WmsgSi1853`
See also:	Option -WmsgSd Option -WmsgSw Option -WmsgSe Pragma MESSAGE

-WmsgSw

+

-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:	With this option a message can be set to a warning message. This option cannot be used in a pragma OPTION. Use the pragma MESSAGE instead.
Example:	`COMPOPTIONS=-WmsgSw2901`
See also:	Option -WmsgSd Option -WmsgSi Option -WmsgSe Pragma MESSAGE

-Wpd

-Wpd: Error for Implicit Parameter Declaration

Group: MESSAGE

Scope: Function

Syntax: "-Wpd".

Arguments: none.

Default: none.

Defines: none.

Pragmas: none.

Description: With this option given the Compiler issues an ERROR message instead of a WARNING message, when an implicit declaration is encountered. This happens if the Compiler does not have a prototype for the called function.

This option helps to prevent parameter passing errors, which will only be detected at runtime. It requires, however, that each function that is called is prototyped before usage. The correct ANSI behavior is to assume that parameters are correct for the call stated.

This option is the same as using `-WmsgSe1801`.

Example: `COMPOPTIONS=-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

-W1

-W1: No Information Messages

Group:	MESSAGE
Scope:	Function
Syntax:	"-W1".
Arguments:	none.
Default:	none.
Defines:	none.
Pragmas:	none.
Description:	Inhibits the Compiler's printing INFORMATION messages, only WARNINGs and ERROR messages are output.
Example:	`COMPOPTIONS=-W1`
See also:	Option -WmsgNi

-W2

-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 printed.
Example:	`COMPOPTIONS=-W2`
See also:	Option -WmsgNi Option -WmsgNw

Compiler Predefined Macros

The ANSI standard for the C language requires the Compiler to predefine a couple of macros. The Compiler provides the following set of predefined macros:

__LINE__ Line number in the current source file.

__FILE__ Name of the source file it appears in.

__DATE__ The date of compilation as a string.

__TIME__ The time of compilation as a string.

__STDC__ Set to 1 if the -Ansi command line option had been given (otherwise additional keywords as far are accepted, which is not in the ANSI standard).

The following table lists all Compiler defines with their names and the option associated with it.

Note that these macros do not have a value, the Compiler treats them as if they had been defined like:

```
#define __HIWARE__
```

It is also possible to log all Compiler predefined defines to a file using the -Ldf option.

Compiler Vendor Defines

The following table shows the defines identifying the Compiler vendor. Note that HIWARE Compilers in the USA may be sold also by ARCHIMEDES.

Name	Defined
`__HIWARE__`	always
`__ARCHIMEDES__`	always

Product Defines

The following table shows the defines identifying the Compiler product. The Compiler may be either a HI-CROSS Compiler (V2.7.x), a Smile~Line Compiler

(V3.0.x) or a HI-CROSS+ Compiler (V5.0.x).

Name	Defined
`__PRODUCT_HICROSS__`	defined for V2.7 Compilers
`__PRODUCT_SMILE_LINE__`	defined for V3.0 Compilers
`__PRODUCT_HICROSS_PLUS__`	defined for V5.0 Compilers
`__VERSION__`	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 how data is organized in memory. There are two possibilities: Little Endian (least significant byte first in memory) or Big Endian (most significant byte first in memory). Usually the ‘Intel World’ uses Little Endian and the ‘Motorola World’ uses Big Endian.

The following two defines are supported:

Name	Defined
`__LITTLE_ENDIAN__`	defined if the Compiler allocates in Little Endian order
`__BIG_ENDIAN__`	defined if the Compiler allocates in Big Endian order

Following example illustrates the difference

```
unsigned long L = 0x87654321;
unsigned short s = *(unsiged short*)&L;  // BE: 0x8765, LE: 0x4321
unsigned char c = *(unsinged char*)&L;   // BE: 0x87,   LE: 0x21
```

Various Defines for Compiler Option Settings

Name	Defined
`__cplusplus`	-C++
`__STDC__`	-Ansi
`__TRIGRAPHS__`	-Ci
`__CNI__`	-Cni
`__OPTIMIZE_FOR_TIME__`	-Ot
`__OPTIMIZE_FOR_SIZE__`	-Os

It is also possible to check in the source if an option is active. The EBNF grammar is

```
OptionActive = "__OPTION_ACTIVE__" "(" string ")".
```

It can be used in the preprocessor and in C code too:

```
#if __OPTION_ACTIVE__("-W2")
  // option -W2 is set
#endif

void main(void) {
  int i;
  if (__OPTION_ACTIVE__("-or")) {
    i=2;
  }
}
```

In the preprocessor, only options valid at this time can be checked.

In C code, it works also with -Odocf and #pragma OPTION.

ANSI-C Standard Types 'size_t' and 'ptrdiff_t' Defines

ANSI provides some standard defines in 'stddef.h' to deal with implementation defined object sizes:

(extract from stdtypes.h (included from stddef.h)):

```
/* size_t: defines the maximum object size type */
  #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 */
  #if defined(__PTRDIFF_T_IS_CHAR__)
    typedef signed char   ptrdiff_t;
  #elif defined(__PTRDIFF_T_IS_SHORT__)
    typedef signed short  ptrdiff_t;
  #elif defined(__PTRDIFF_T_IS_INT__)
    typedef signed int    ptrdiff_t;
  #elif defined(__PTRDIFF_T_IS_LONG__)
    typedef signed long   ptrdiff_t;
  #else
    #error "illegal ptrdiff_t type"
  #endif
```

The following defines are used to deal with different possible implementations:

Macro	Description
`__SIZE_T_IS_UCHAR__`	Defined if the Compiler expects size_t in stddef.h to be 'unsigned char'.
`__SIZE_T_IS_USHORT__`	Defined if the Compiler expects size_t in stddef.h to be 'unsigned short'.
`__SIZE_T_IS_UINT__`	Defined if the Compiler expects size_t in stddef.h to be 'unsigned int'.
`__SIZE_T_IS_ULONG__`	Defined if the Compiler expects size_t in stddef.h to be 'unsigned long'.
`__PTRDIFF_T_IS_CHAR__`	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'char'.
`__PTRDIFF_T_IS_SHORT__`	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'short'.
`__PTRDIFF_T_IS_INT__`	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'int'.

Macro	Description
`__PTRDIFF_T_IS_LONG__`	Defined if the Compiler expects ptrdiff_t in stddef.h to be 'long'

The following tables inform about the default settings of the for the ANSI-C Compiler standard types size_t and ptrdiff_t:

Macros for HC08

The following tables shows the settings for this target:

size_t Macro	Defined
__SIZE_T_IS_UCHAR__	never
__SIZE_T_IS_USHORT__	never
__SIZE_T_IS_UINT__	always
__SIZE_T_IS_ULONG__	never

ptrdiff_t Macro	Defined
__PTRDIFF_T_IS_CHAR__	never
__PTRDIFF_T_IS_SHORT__	never
__PTRDIFF_T_IS_INT__	always
__PTRDIFF_T_IS_LONG__	never

Object File Format Defines

Depending on the specified object file format option, the Compiler defines some macros to identify the format (mainly used in the startup code, because it may be object file specific). Note that the options -F1 and -F2 are only available in HI-

CROSS+.

Name	Defined
`__HIWARE_OBJECT_FILE_FORMAT__`	-Fh
`__ELF_OBJECT_FILE_FORMAT__`	-F1, -F2

Bit Field Allocation Defines

The Compiler provides four 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 */
```

With this defines it is possible to write compatible code over different Compiler vendors even if the bit field allocation differs and allocation order of bit fields is important:

```
    struct {
      /* Memory layout of I/O port:
                     MSB                          LSB
        name:        BITA | CCR | DIR | DATA | DDR2
        size:        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;
```

Note that if the basic allocation unit for bitfields in the Compiler is a byte, allocation of memory for bitfields always is from most significant BYTE to least significant BYTE, thus __BITFIELD_MSBYTE_FIRST__ may be defined always for such a Compiler:

```
/* example for __BITFIELD_MSBYTE_FIRST__ */
struct {              /* LSBIT_FIRST       */  /* MSBIT_FIRST       */
  unsigned char a:8; /* MSByte   LSByte   */  /* MSByte   LSByte   */
  unsigned char b:3; /* aaaaaaaa cccccbbb */  /* aaaaaaaa bbbccccc */
  unsigned char c:5;
} MyIOport2;
```

Note: There is no standard defined way to allocating bit fields. Therefore allocation may vary from compiler to compiler even for the same target. Using bit fields for access to I/O registers is thus highly 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.

Macros for HC08

The following table identifies the implementation in the Back End.

Name	Defined
__BITFIELD_MSBIT_FIRST__	-BfaBMS
__BITFIELD_LSBIT_FIRST__	-BfaBLS
__BITFIELD_MSBYTE_FIRST__	always
__BITFIELD_LSBYTE_FIRST__	never

Type Information Defines

The Flexible Type Management (Compiler Option -T, only available with HI-

CROSS+) sets some defines to identify the type sizes.

Name	Defined
__CHAR_IS_SIGNED__	see option -T or Back End
__CHAR_IS_UNSIGNED__	see option -T or Back End
__CHAR_IS_8BIT__	see option -T or Back End
__CHAR_IS_16BIT__	see option -T or Back End
__CHAR_IS_32BIT__	see option -T or Back End
__CHAR_IS_64BIT__	see option -T or Back End
__SHORT_IS_8BIT__	see option -T or Back End
__SHORT_IS_16BIT__	see option -T or Back End
__SHORT_IS_32BIT__	see option -T or Back End
__SHORT_IS_64BIT__	see option -T or Back End
__INT_IS_8BIT__	see option -T or Back End
__INT_IS_16BIT__	see option -T or Back End
__INT_IS_32BIT__	see option -T or Back End
__INT_IS_64BIT__	see option -T or Back End
__ENUM_IS_8BIT__	see option -T or Back End
__ENUM_IS_SIGNED__	see option -T or Back End
__ENUM_IS_UNSIGNED__	see option -T or Back End
__ENUM_IS_16BIT__	see option -T or Back End
__ENUM_IS_32BIT__	see option -T or Back End
__ENUM_IS_64BIT__	see option -T or Back End
__LONG_IS_8BIT__	see option -T or Back End
__LONG_IS_16BIT__	see option -T or Back End
__LONG_IS_32BIT__	see option -T or Back End
__LONG_IS_64BIT__	see option -T or Back End
__LONG_LONG_IS_8BIT__	see option -T or Back End
__LONG_LONG_IS_16BIT__	see option -T or Back End
__LONG_LONG_IS_32BIT__	see option -T or Back End

Name	Defined
__LONG_LONG_IS_64BIT__	see option -T or Back End
__FLOAT_IS_IEEE32__	see option -T or Back End
__FLOAT_IS_IEEE64__	see option -T or Back End
__FLOAT_IS_DSP__	see option -T or Back End
__DOUBLE_IS_IEEE32__	see option -T or Back End
__DOUBLE_IS_IEEE64__	see option -T or Back End
__DOUBLE_IS_DSP__	see option -T or Back End
__LONG_DOUBLE_IS_IEEE32__	see option -T or Back End
__LONG_DOUBLE_IS_IEEE64__	see option -T or Back End
__LONG_DOUBLE_IS_DSP__	see option -T or Back End
__LONG_LONG_DOUBLE_IS_IEEE32__	see option -T or Back End
__LONG_LONG_DOUBLE_IS_IEEE64__	see option -T or Back End
__LONG_LONG_DOUBLE_IS_DSP__	see option -T or Back End
__VTAB_DELTA_IS_8BIT__	see option -T
__VTAB_DELTA_IS_16BIT__	see option -T
__VTAB_DELTA_IS_32BIT__	see option -T
__VTAB_DELTA_IS_64BIT__	see option -T

Motorola HC08 Specific Defines

The following table identifies specific implementations in the Back End.

Name	Defined
__HC08__	always
__SMALL__	-Ms
__BANKED__	-Mb
__PTR_SIZE_1__	never
__PTR_SIZE_2__	always

Name	Defined
`__PTR_SIZE_3__`	never
`__PTR_SIZE_4__`	never

Compiler Pragmas

A pragma is a way of passing information from the front end to the Back End without affecting the parser. In the Compiler the effect of a pragma onto 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 `ONCE` and `NO_STRING_CONSTR` which are valid for one file.

The syntax of a pragma is:

```
#pragma pragma_name [optional_arguments]
```

The format of the optional arguments depends on the pragma's name – some pragmas don't have any arguments at all.

Note: A pragma directive accepts one single pragma with optional arguments. Do not put more than one pragma in a pragma directive as in the following illegal example: #pragma ONCE NO_STRING_CONSTR

*Note: This is **illegal**!*

All pragmas affecting the Front End are described below. All other pragmas only affect the code generation process and therefore are described in the Back End section.

Pragma Details

The remainder of this section is devoted to describing each of the pragmas available for the Compiler. The pragmas are listed in alphabetical order and each is divided into several sections.

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.

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.

The following table gives an overview of the different scopes of pragmas.

Scope	Description
File	The pragma is valid from the current position until the end of the source file. E.g. 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 hole compilation unit. E.g. 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. Be careful that always a data definition is behind this pragma in a header file. If not, the pragma is used for the first data in next header file or in the main file.
Function Definition	The pragma affects only the next function definition. Be careful if such a pragma is used 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 following this one, it is valid until the end of the file.

#pragma CODE_SEG

#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,NEAR, CODE, FAR, BIT, Y_BASED, Z_BASED, DPAGE, PPAGEEPAGE

The meaning of these segment modifiers are backend dependent. Please 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 is used to specify the segment a function is allocated in. The segment modifiers do also specify the calling convention of a function.
The pragma CODE_SEG sets the current code segment. This segment is used to place all new function definitions. Also all function declarations get the current code segment when they occur. The segment modifiers of this segment are then used to determine the calling convention.

The pragma CODE_SEG does affect function declarations, not only definitions. It should be carefully designed that all function declarations and their definition 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 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 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 backends do support a FAR calling convention.

Note: The calling convention can also be specified with a keyword, is supported and additionally, the default calling convention can be 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 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 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 CONST_SEG

#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 :

DIRECT,NEAR, CODE, FAR, BIT, Y_BASED, Z_BASED, DPAGE, PPAGEEPAGE

The meaning of these segment modifiers are backend dependent. Please read the backend chapter to find the supported modifiers and their meaning. The SHORT modifier is usually used to specify a segment which can be accesses with 8 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 is used to allocate constant variables into a segment. Then this segment is located in the link parameter file to specific addresses.
The pragma CONST_SEG sets the current const segment. This segment is used to place all constant variable declarations. The default segment can be set with

```
#pragma CONST_SEG DEFAULT”
```

Constants are allocated in the current data segment defined with #pragma DATA_SEG in the HICROSS object file format when the option -cc is not specified until the first #pragma CONST_SEG occurs in the source. With option -Cc, in the ELF object file format and after the first #pragma CONST_SEG, constants are always allocated in constant segments.

The pragma CONST_SEG does also affect constant variable dec-

larations, not only definitions. It should be carefully designed that all constant variable declarations and the constant variable definitions are in the same const segment.

Note that some compiler optimizations do assume that objects having the same segment are together. So 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. So when using a paged access to variables, put 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:

```
/* in a header file */
#pragma CONST_SEG SHORT 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 SHORT_CONST_MEMORY
const int i_short=7;
#pragma CONST_SEG CUSTOM_CONST_MEMORY
const int j_custom=8;
#pragma CONST_SEG DEFAULT
```

Illegal usages of the pragma

```
#pragma DATA_SEG CONST1
#pragma CONST_SEG CONST1
/* error: segment name has different types ! */
```

```
#pragma CONST_SEG CONST1
#pragma CONST_SEG SHORT CONST1
/* 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 DEFAULT
/* error: modifiers for DEFAULT segment no 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 CREATE_ASM_LISTING

#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.

Actually an 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 accesible from assmembler code*/
#pragma CREATE_ASM_LISTING OFF
extern int j; /* j is only accesible from C code */
```

See also: Compiler Option -La
Generating an Assembler Include File

#pragma DATA_SEG

#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 :

DIRECT,NEAR, CODE, FAR, BIT, Y_BASED, Z_BASED, DPAGE, PPAGEEPAGE

The meaning of these segment modifiers are backend dependent. Please read the backend chapter to find the supported modifiers and their meaning. The SHORT modifier is usually used to specify a segment which can be accesses with 8 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 is used to allocate variables into a segment. Then this segment is 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 can be set with

```
#pragma DATA_SEG DEFAULT"
```

Constants are also allocated in the current data segment in the HICROSS object file format when the option -cc is not specified and no "#pragma CONST_SEG" did occur in the source so far. With option -Cc and in the ELF object file format, constants are not allocated in the data segment.

The pragma DATA_SEG does also affect data declarations, not only definitions. It should be carefully designed that all variable declarations and the variable definitions are in the same segment.

Note that some compiler optimizations do assume that objects having the same segment are together. So 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. So 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:

```
/* in a header file */
#pragma DATA_SEG SHORT 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 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;
}
```

Illegal usages of the pragma

```
#pragma DATA_SEG DATA1
#pragma CONST_SEG DATA1
/* error: segment name has different types ! */

#pragma DATA_SEG DATA1
#pragma DATA_SEG SHORT 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 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 INLINE

#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 if possible.

The pragma is the same as using the ‘inline’ keyword in C++ or using the -Oi Compiler option.

Example:

```
int i;
#pragma INLINE
static void foo(void) {
  i = 12;
}

void main(void) {
  foo();  // results into ‘i = 12;’
}
```

See also: Compiler Option -Oi
C++ Front End

#pragma INTO_ROM

#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 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) also disables the pragma.

Example:

```
#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

#pragma LOOP_UNROLL

#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 MESSAGE

#pragma MESSAGE: Message Setting

Scope: Compilation Unit

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: With this pragma messages can selectively be set to a information, a warning, disable or to an error.

Example: In the following example braces () were forgotten. The code without the braces is still legal C, but it is very probably what the programer did intend.

```
/* 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

#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. This may be useful for inline assembler functions.

The code generated in a function with #pragma NO_ENTRY may not be save! It is assumed that the user takes care about stack usage himself.

Note: Not all backends do support this pragma. Some do still generate entry code even if this pragma is specified.

Example: Avoid any generated code by the compiler. All code is written in inline assembler.

```
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_FRAME
#pragma NO_RETURN
void Func0(void) {
  asm {/* no code should be writen by the compiler.*/
    ...
  }
}
```

See also: #pragma NO_EXIT
#pragma NO_FRAME
#pragma NO_RETURN

#pragma NO_EXIT

#pragma NO_EXIT: No Exit Code

Scope: Function

Syntax: "#pragma NO_EXIT".

Synonym: none.

Arguments: none.

Default: none.

Description: This pragma suppresses the generation of the exit code. This may be useful for inline assembler functions.

The code generated in a function with #pragma NO_ENTRY may not be save! It is assumed that the user takes care about stack usage himself.

Note: Not all backends do support this pragma. Some do still generate exit code even if this pragma is specified.

Example: Avoid any generated code by the compiler. All code is written in inline assembler.

```
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_FRAME
#pragma NO_RETURN
void Func0(void) {
  asm {/* no code should be writen by the compiler.*/
    ...
  }
}
```

See also: #pragma NO_ENTRY
#pragma NO_FRAME
#pragma NO_RETURN

#pragma NO_FRAME

#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. This may be useful for inline assembler functions.

The code generated in a function with #pragma NO_ENTRY may not be save! It is assumed that the user takes care about stack usage himself.

Note: Not all backends do support this pragma. Some do still generate entry code even if this pragma is specified.

Example: Avoid any generated code by the compiler. All code is written in inline assembler.

```
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_FRAME
#pragma NO_RETURN
void Func0(void) {
  asm {/* no code should be writen by the compiler.*/
    ...
  }
}
```

See also: #pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_RETURN

#pragma NO_LOOP_UNROLL

#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

#pragma NO_RETURN: No Return Instruction

Scope: Function

Syntax: "#pragma NO_RETURN".

Synonym: none.

Arguments: none.

Default: none.

Description: This pragma suppresses the generation of the return instruction (return from subroutine, return from interrupt). This may be useful if the programmer itself cares 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), it just suppresses the generation of the return instruction.

Note: 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 save side allocate both function into a segment, which does only have a linear memory area.

Example: The following example places some functions into a special named segment. All functions in this special code segment have to be called from a operating system every 2 seconds each after each other. With the pragma, some functions does not return and fall directly to the next function to be called, saving code size and execution time:

```
#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

#pragma NO_STRING_CONSTR: No String Concatenation during Preprocessing

Scope: Compilation Unit

Syntax: "#pragma NO_STRING_CONSTR".

Synonym: none.

Arguments: none.

Default: none.

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, which can be very useful if a macro is to contain inline assembler statements using this character, e.g. for IMMEDIATE values.

Example: The following pseudo assembly code macro shows the usage of the pragma. Without the pragma '#' would be 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

#pragma ONCE: Include Once

Scope:	File
Syntax:	"#pragma ONCE".
Synonym:	none.
Arguments:	none.
Default:	none.
Description:	If this pragma appears in a header file, this file is opened and read only once, which can speed up compilation tremendously.
Example:	#pragma ONCE
See also:	Option -Pio

#pragma OPTION

#pragma OPTION: Additional Options

Scope: Compilation Unit

Syntax: "#pragma" "OPTION" ("ADD" [<Handle>] {<Option>} |"DEL" ({"Handle"} | "ALL")).

Synonym: none.

Arguments: <Handle> : A identifier, so that added options can selectively be deleted again.

<Option> : A valid option string enclosed in " characters.

Default: none.

Description: With this pragma options can be 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.

With the “ADD” command, additional options can be added to the current ones. A handle may be given optionally.

The “DEL” command can either remove all options with a specific handle or with the “ALL” keyword all added options regardless if they have a handle or not.

All keywords and also 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 can be 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 following example shows how to compile only a single function with the additional option -or.

```
#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 */
```

Here some illegal examples :

```
#pragma OPTION ADD -or /* ERROR, use “-or” */
#pragma OPTION  “-or” /* ERROR, use keyword ADD */
#pragma OPTION ADD “-odocf=\”-or\””
/* ERROR, use “-odocf” not allowed in this pragma */

void f(void) {
#pragma OPTION ADD “-or”
/* ERROR, pragma not allowed inside of declarations */
}
#pragma OPTION ADD “-cni”
#ifdef __CNI__
/* ERROR, macros are not defined for options */
/* added with the pragma */
#endif
```

See also: none.

#pragma STRING_SEG

#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,NEAR, CODE, FAR, BIT, Y_BASED, Z_BASED, DPAGE, PPAGEEPAGE

The meaning of these segment modifiers are backend dependent. Please read the backend chapter to find the supported modifiers and their meaning. The SHORT modifier is usually used to specify a segment which can be accesses with 8 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 is used to allocate strings into a segment. By default, strings are allocated in the linker segment STRINGS. With this pragma, strings can alternatively allocated in special segments. Moreover string segments may have modifiers, so that the compiler knows how 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 can be 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 6 bytes. When putting strings into user defined segments, the linker may no longer do this optimization. So only use

a user defined string segment 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 TRAP_PROC

#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), because 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. In this case 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, the function can be declared with 'extern "C"'
Examples:
extern "C" interrupt MyResetFkt(void) { ...
or
#pragma TRAP_PROC
extern "C" MyResetFkt(void) { ...

Example:

```
#pragma TRAP_PROC
void MyInterrupt(void) {
  ...
}
```

See also: interrupt keyword

ANSI-C Front End

The front end of the Compiler reads the source file(s), does all the syntactic and semantic checking and produces some intermediate representation of the program which then is passed on to the Back End to generate code from.

This section discusses features, restrictions and further properties of the ANSI-C front end of the Compiler.

Implementation Features

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. Instead of introducing additions to the language to support features such as interrupt procedures, the Compiler provides a series of pragmas.

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.The line directive is ignored by the Compiler.

Language Extensions

There is a language extension in the Compiler for ANSI-C: Keywords can be used

to qualify pointers in order to distinguish them or to mark interrupt routines.

The Compiler may support the following non-ANSI compliant keywords (see Back End if they are supported and for their semantic):

```
far
near
rom
uni
interrupt
@address
```

Note: Not all Back Ends may support those keywords. See section "Non-ANSI Keywords" in the Back End.

ANSI-C was not designed with embedded controllers in mind. Those keywords are not ANSI conform, but they allow a easy way to get best results in the embedded application world.

The interrupt keyword may be used to mark functions as interrupt functions and to link the function to a specified interrupt vector number. Far, near, rom and uni may be used to distinguish between different pointer types (e.g. for paging). Far and near may be used also to specify different calling conventions (e.g. if banking is available).

To allow portable programming between different CPUs or if the target CPU does not support an additional keyword, the header file 'hidef.h' may contain some defines as below:

```
#define far    /* no far keyword supported */
#define near   /* no near keyword supported */
```

Global variable address modifier (@address)

Global variables can be put to specific addresses with the global variable address modifier. This is useful for accessing memory mapped I/O ports.

It has following syntax:

```
Declaration = <TypeSpec> <Declarator>
              [@ <Address>] [= <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.

Take care about the choice of the address: For each global object specified with an absolute address a segment is created. Thus 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 for addressing the default data page, pointers pointing to this global variable must be "far". An alternative way to put global variables to specific addresses is the use of

```
#pragma DATA_SEG [SHORT] <segment_name>
```

with setting the PLACEMENT section in the linker parameter file

```
<segment_name> INTO  READ_ONLY <Address> ;
```

This was previously the only way to do it.

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
```

far Keyword

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. It might be used e.g. to access memory mapped

I/O registers that are not on the data page, or one might choose to allocate constant strings in a ROM not on the data page.

For function ‘far’ defines the calling convention for this function. Some backends support special calling conventions which do also set a page register when a function is called. This way more code than the usual address space can be used. Consider that the special allocation of such functions in not done automatically.

Usage of far for Pointers

The keyword `far` is a type qualifier like `const` and is only valid in the context of pointer types and functions. For pointers always affects the last '*' to its left in a type definition, i.e. the declaration of a `far` pointer to a `far` pointer to a character looks like this:

```
char *far *far p;
```

and 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 write the `far` *keyword after the asterisk, i.e. write*

```
char *far p;
```

and not

```
char far *p;
```

The second possibility is illegal!

far and arrays

There is one case where the `far` keyword does not appear in the context of the '*' type constructor: in the declaration of an array parameter:

```
void my_func (char a[37]);
```

Of course, such a declaration actually specifies a pointer argument, so this is equal to

```
void my_func (char *a);
```

To declare such an argument as a far pointer, there are the following two possibilities:

```
void my_func (char a[37] far);
```

or alternatively

```
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, e.g. 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 looks like this:

```
void my_func (ARRAY far a);
```

In this case, the parameter is actually a `far` pointer to the first element of the array, i.e. 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);
```

but it is *not* the same as specifying a `far` pointer to the array:

```
void my_func (ARRAY *far a);
```

because now `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 the same addressing mode for this variable as it has been declared in a FAR segment. Note that for the above variable declarations/definitions the variable are in the DEFAULT_DATA segment if no other data segment is active, so be careful if you mix ‘far’ declarations/definitions within a non-FAR data segment. Assuming that FAR segments have ‘extended’ addressing mode and normal segments have ‘direct’ addressing mode, the following two examples may clarify this behavior:

```
// ok, consistent declartions
#pragma DATA_SEG MyDirectSeg  // use direct addressing mode
int i;     // direct, segment MyDirectSeg
```

```
int j;     // direct, segment MyDirectSeg
#pragma DATA_SEG FAR 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
```

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 of the owner class of the function is not allocated on the default data page. Example:

```
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 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 modifier `far` can be applied to references. This is useful, if it is a reference to an object outside of the default data page. 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;
};
```

Usage of far for Functions

With the far keyword, also a special calling convention can be specified. In this context, 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 star (“*”).

```
int far *f(void);
```

It must, however be after the int and not before it.

For function pointers many backends are using assumption that far function pointer are pointing to functions with far calling convention, even if the calling convention was not specified. Moreover most of the backends do not support different function pointer sizes in one compilation unit. The function pointer size then is then only dependent from the memory model. See the backend chapter for details:

declaration	allowed	type description
int far f();	ok	far function returning an int
far int f();	error	
far f();	ok	far function returning an int
int far *f();	ok	far function returning a pointer to int
int * far f();	ok	function returning a far pointer to int
far int * f();	error	
int far * far f();	ok	far function returning a far pointer to int
int far i;	ok	global far object
int far *i;	ok	pointer to a far object
int * far i;	ok	far pointer to int
int far * far i;	ok	far pointer to a far object
far int *i;	ok	pointer to a far integer
int *far (* far f)(void)	ok	far pointer to function returning a far pointer to int
void * far (* f)(void)	ok	pointer to function returning a far pointer to void
void far * (* f)(void)	ok	pointer to far function returning a pointer to void

near Keyword

Note: Not all Back Ends may support this keyword. See section "Non-ANSI Keywords" in the Back End.

From the point of the language, the near keyword can anywhere by used instead of the far keyword. It is used in situations where the far modifier is the default and especially short and objects should be specified, or where the near calling convention should be explicitly specified.

As the far keyword is has two semantics. Either it specifies a small size of function or data pointers or it specifies the near calling convention.

declaration	allowed	type description
int near f();	ok	near function returning an int
int near far f();	error	
near f();	ok	near function returning an int
int near * far f();	ok	near function returning a far pointer to int
int far *i;	error	
int * near i;	ok	far pointer to int
int * far* near i;	ok	near pointer to far pointer to int
int *far (* near f)(void)	ok	near pointer to function returning a far pointer to int
void * near (* f)(void)	ok	pointer to function returning a near pointer to void
void far *near (*near f)(void)	ok	near pointer to far function returning a far pointer to void

rom Keyword

Note: Not all Back Ends may support this keyword. See section "Non-ANSI Keywords" in the Back End.

Some micro controllers support a Harvard Architecture instead the Von Neumann Architecture.

In the Von Neumann Architecture code (ROM) and data (RAM) are accessible with the same load/store instructions, but in the Harvard Architecture ROM and RAM are separate address spaces and special instructions have to be used.

Using a a Harvard Architecture values stored in ROM can only be accessed using

special instructions. A `rom` pointer always points to ROM. `rom` pointers may be used to access string constants located in ROM.

Example:

```
const char * rom const ptr; /* points to code; ptr in ROM */
```

uni Keyword

Note: Not all Back Ends may support this keyword. See section "Non-ANSI Keywords" in the Back End.

When using a Harvard Architecture values stored in ROM can only be accessed using the `MOVC` instruction. Therefore, a pointer must contain the information which kind of memory it points to (RAM or ROM). This information must be present when working with pointers, because a pointer can be copied at run time and the handling function would not know whether this pointer refers to the code or data memory. Therefore, an additional pointer type has to be introduced: the `uni` (universal) pointer. Such `uni` pointers have an additional byte which signals the kind of memory to be accessed. To handle memory accesses by universal pointers, run time functions may be used.

Example:

```
const char * uni ptr;
/* points to code; ptr in RAM */
char  * const ptr;
/* points to data; ptr in ROM */
  const char * uni const ptr;
/* points to code; ptr in ROM */
```

For global initialization, the correct pointer type has to be specified, e.g. it may be dangerous to use uni pointers for string pointer allocation, if the strings are allocated in a rom section:

```
const char *const uni Bad[] = {"abc", "def"};  // wrong, use 'rom'
const char *const rom Ok[] = {"abc", "def"};  // ok, rom pointers
```

Compatibility

Far pointers and normal pointers are compatible. If necessary, the normal pointer is extended with the default data page to a far pointer (subtraction of two pointers or assignment to a far pointer). In the other case, the far pointer simply is clipped to a normal pointer, i.e. the page part is discarded.

interrupt Keyword

Note: Not all Back Ends may support this keyword. See section "Non-ANSI Keywords" in the Back End.

To specify a function to be an interrupt routine can be done in 2 different ways:

1. by using the #pragma TRAP_PROC and adapting the Linker parameter file.

2. by using the non-standard interrupt keyword.

The non-standard interrupt keyword can be used like any other type qualifier. It specifies a function to be an interrupt routine. It can be followed by a number specifying the entry in the interrupt vector, that should contain the address of the interrupt routine. If 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, but the number of the entry in the vector has to be specified in the Linker parameter file with the Linker command `VECTOR` number functionname.

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()

void h() interrupt; // ok
  // same as #pragma TRAP_PROC,
  // please set the number in prm-file

int i() interrupt 4; // ok
  // fourth entry in vector points to i()

int i() interrupt 5; // error, redeclaration with
  // other number than first declaration

interrupt 4 int j() interrupt 3;
  // error, twice interrupt

interrupt 3 interrupt 4 k();
  // error, twice interrupt

interrupt int l; // error, not a function
```

Example:

```
interrupt 2 int g();
// The second entry (number 2) gets the address of function g().
```

Implementation Defined Behavior

The ANSI standard contains a couple of places where the behavior of a particular Compiler is left undefined. Thus it is possible for different Compilers to implement certain features in different ways, even if they all comply to the ANSI-C standard. Subsequently, we'll discuss those points and the behavior implemented by the Compiler.

Right Shifts

For right shift of an object with signed type having a negative value, the result of E1 >> E2 is implementation-defined, 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

In ANSI-C not allowed! We allow 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 Back End chapter about default settings.

Division and Modulus

The results of the "/" and "%" operators also are not properly defined for signed arithmetic 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 Back End for details and examples.

Translation Limitations

This section describes the internal limitations of the Compiler

Limitation	Description
Block Nesting	The nesting level of blocks (blocks are braced by '{' and '}') must not exceed 32.
Structs & Unions	The nesting level in a `struct` or `union` definition must not exceed 32 either.
Type Declarations	Derived types must not contain more than 100 components.
Cases in Switches	There may be at most 1000 `case` labels in a `switch` statement.
Labels	There may be at most 16 other labels within one procedure.
Macro Expansion	Expansion of macros is limited to 70 recursive expansions.
Include Files	The number of include files is limited to 8196 for a single compilation unit.
Numbers	Maximum of 10’000 different number for a single compilation unit (message C2700, message C3302).
Goto	Maximum of 128 gotos for a single function.
Parsing Recursion	Maximum of 200 parsing recursions.
Expression Recursion	Maximum of 32 recursion level for expressions (message C4006)
String Concatenation	Maximum length for string concatenation during preprocessing are 2048 characters (message C3301)
Include File Recursion	Maximum depth for include file hierarchy is 32 in a single compilation unit (message C3000).
Lexical Tokens	Maximum of 512000 lexical source tokens (message C3200).
Identifier Length	A identifier must not exceed 32k characters.
Internal ID’s	Maximum of 256 internal ID’s for a single compilation unit (message C3304). Internal ID’s are used for additional local/global variables created by the Compiler (e.g. by using CSE).
String Length	Maximum length for strings during preprocessing (implicit string concatenation, include file names) are 2048 characters (message C3301, message C4408).
Macros	Maximum of 10’000 macros for a single compilation unit (message C4403).

Limitation	Description
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

Because many programmers have problems with the conversion rules in ANSI-C, this section gives a short overview about the implementation (see also ANSI Standard 6.2)

Integral promotions

A char, a short int, or an int bit-field, or their signed or unsigned varieties, or an enum type, may be used in an expression wherever an int or unsigned int may be used. If an int can represent 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 filling the high order bits with copies of the sign bit.

Converting a signed integer type to an unsigned inter type: If the destination has equal or greater size, first signed extension of the signed integer type is done. If the destination has smaller size, the result is the remainder on division by the number one greater than the largest unsigned number that can be represented in the type with 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.
- Otherwise, if either operand has type double, the other operand is converted to double.
- Otherwise, if either operand has type float, the other operand is converted to float.
- Otherwise, 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.
- Otherwise, 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.
- Otherwise, if either operand has type long int, the other operand is converted to long int.
- Otherwise, if either operand has type unsigned int, the other operand is converted to unsigned int.
- Otherwise, 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.
```

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

In general, the Compiler supports two IEEE floating point formats: IEEE32 and IEEE64. Additionally there may be a DSP format supported by the processor. Next Figure shows these three formats.

IEEE 32bit Format (Precision: 6.5 decimal digits)

	8bit exp	23 bit mantissa

sign bit

value = $-1^s * 2^{(e-127)} * 1.m$

IEEE 64bit Format (Precision: 15 decimal digits)

	11bit exp	52 bit mantissa

sign bit

value = $-1^s * 2^{(e-1023)} * 1.m$

DSP Format (Precision: 4.5 decimal digits)

16bit mantissa	16bit exponent

value = $m * 2^e$ (no hidden bit)

Negative exponents are in 2’s complement, the mantissa in signed fixed-point format.

Normally, floats are implemented as IEEE32 and doubles as IEEE64, but this may vary for a specific Back End or not both formats may 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 :

value = (-1)^s * m*2^e
where sign is 0 or 1,
2 > m >= 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).
```

Notice because this number is computed to be larger or equal to 1 and smaller than 2, in front of the point there is always a 1. For the remaining steps this constant 1 is left away 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)
```

So the fields must only be put together according to the graphic:

```
500.0 (dec)
= 0 (sign) 10000111 (exponent)
     11110100000000000000000 (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)
     11110100000000000000000 (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.

```
500.0 (dec)
= 0 (sign) 10000000111 (exponent)
    1111010000000000000000000000000000000000000000000000 (mantissa)
    (IEEE64 as bin)
= 0100 0000 0111 1111 0100 0000 0000 0000 0000 0000 0000
    0000 0000 0000 0000 0000 (IEEE64 as bin)
= 40 7f 40 00 00 00 00 00 (IEEE64 as hex)

-500.0 (dec)
= 1 (sign) 10000000111 (exponent)
    1111010000000000000000000000000000000000000000000000 (mantissa)
    (IEEE64 as bin)
= 1100 0000 0111 1111 0100 0000 0000 0000 0000 0000 0000
    0000 0000 0000 0000 0000 (IEEE64 as bin)
= c0 7f 40 00 00 00 00 00 (IEEE64 as hex)
```

Note: The IEEE formats knows several special bit patterns for special values. The number 0 (zero) is encoded by the bit pattern consisting of zero bits only. Also other special values like “Not a number”, “infinity”, -0 (minus zero) and denormalized numbers do exist. Please refer to the IEEE standard documentation for details.
Except the 0 (zero) and -0 (minus zero) special formats not all special forms may be supported for specific backends.

Representation of 500.0 in DSP format

First 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 represent also 0, which must have a special patters 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).
```

Notice because this number is computed to be always larger or equal to 0 and smaller than 1, in front of the point there is always a 0. For the remaining steps this constant is left away to save space. After the decimal point there is always a 1, except for 0 and intermediate results. This bit is encoded, so DSP looses 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)
```

For negative numbers, the twos complement is taken :

```
mant(-500.0, DSP) = 1000001100000000 (bin)
```

Finally the mantissa end 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 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 most significant byte.

Volatile Objects, Absolute Variables

The Compiler does not do register- and constant tracing on volatile/absolute global objects. Thus it is guaranteed that 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 defined way to allocate bit fields. Therefore allocation may vary from Compiler to Compiler even for the same target. Using bit fields for access to I/O registers is thus highly 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.

The maximum width of bit fields is Back End dependent (see Back End for details), 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

When using 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, where this mistake is made:

```
typedef struct _B {
```

```
  signed int b0: 1;
} B;

B b;

if (b.b0 == 1) …
```

The Compiler will issue a warning and replace the 1 with -1, because the condition `(b.b0 == 1)` does not make sense, i.e. it is always false. Then the test `(b.b0 == -1)` is performed, as the user is expecting. This substitution is not ANSI compatible and it will not be performed when the –Ansi Compiler option is active.

The correct way to specify this example 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, it can have the values 0 or 1, and thus the test `(b.b0 == 1)` is correct and does make sense.

Recommendations

In order to save memory, it may be a good idea to implement globally accessible boolean flags as unsigned bit fields of width 1. However, we do 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, because different Compilers may allocate bit fields differently.

For information about how the Compiler allocates bit fields, see section *Back End, Data Types*.

Segmentation

The Linker supports the concept of segments, i.e. 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" | "NEAR" | "CODE" | "FAR" | "BIT" | "Y_BASED" |
            "Z_BASED" | "DPAGE" | "PPAGE" | "EPAGE".
```

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 pragma 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 HICROSS 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 was specified. After this pragma, all strings are allocated in 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). To explicitly make these default segments the current ones, use the segment name `DEFAULT:`

```
#pragma CODE_SEG DEFAULT
#pragma DATA_SEG DEFAULT
#pragma CONST_SEG DEFAULT
#pragma STRING_SEG DEFAULT
```

Segments may also be declared as `SHORT` by inserting the keyword `SHORT` just before the segment name (with the exception of predefined segment `DEFAULT` – this segment cannot be `SHORT`). 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 in such a segment. Of course it’s the programmer's responsibility to actually allocate `SHORT` segments in the proper memory area!

Note: The default code and data segments may not be declared `SHORT`*.*

The meaning of the other segment modifiers like NEAR, FAR, ... are backend specific. Modifiers a backend does not support, are ignored. Please see in the backend chapter about which modifiers are supported.

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: For all segment names, there are aliases to satisfy the ELF naming conventions. Instead of CODE_SEG also CODE_SECTION, instead of DATA_SEG also DATA_SECTION, instead of CONST_SEG also CONST_SECTION and instead of STRING_SEG also STRING_SECTION is allowed. These aliases behave exactly are the XXX_SEG name versions.

Example for Segmentation **Without** Compiler Option -Cc

```
                                /* 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 Option -Cc

```
                                   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 commonly subsumed under the term "optimization". This section gives a short overview about the most important optimizations.

Peephole Optimizer

A peephole optimizer is a simple optimizer in a Compiler. This 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 a highly optimizing Compiler, it is still possible that code patterns may result that can be further 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 do only make sense in conjunction with other optimizations or together with some code patterns. These patterns may have been generated by doing other optimizations first. 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 can be either execution time or code size. Examples are the replacement of multiplications and divisions by constant powers of two with left or right shifts.

Note: The compiler can only replace a division by two by 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. Otherwise 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 analyzed intensively. 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, or in other words: 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 and 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 will need 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, but their use in expressions are replaced by the constant value they hold, too.

Constant non-volatile arrays are optimized, too.

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:

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 number.

Method	Description
Branch Sequence	For small switches with scattered case label values the Compiler generates a if ... elsif ... elsif ... else ... sequence if the Compiler switch -Os is active.

Method	Description
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 anymore 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 like

```
float x, y;
x = (y < 0.0) ? -y : y;
```

which results in huge and inefficient code. The Compiler recognizes cases like this one 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 statements

```
x = x + 1;
x += 1;
x++;
```

and even that between

```
x = x / y;
x /= y;
```

Therefore, the Compiler can generate equally efficient code for either case.

Usage of Qualifiers for Pointers

For not so experienced programmers, the usage of qualifiers (const, volatile, ...) for pointers seems not so clear. The motivation of this section is to give some examples for the usage of const/volatile, because const and volatile are very common for Embedded Programming.

First a simple example:

```
int i;
const int ci;
```

The above definitions are a ‘normal’ variable ‘i’ and a constant variable ‘ci’ which can be placed into ROM. Note that for C++ the constant ‘ci’ has to 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’.

Confused? Maybe it helps if you know that the qualifier for such pointers is always on the right side of a ‘*’. Another way is just to read the source from right to left.

Now what about a ‘array of five constant pointers to volatile integers’? Because the same rule apply to volatile too, this is easy to express:

```
volatile int *const arr[5];
```

Is this array placed into ROM? ‘arr’ is an array of five constant pointers pointing to volatile integers. Because the array itself is constant, it can be in ROM. Where the pointers point to does not matter if the array is constant or not. But what about the next example?

```
const char *const *buf[] = {&a, &b};
```

Even 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.

Now something complicated: What about a constant array of five ordinary function pointers? With the knowledge that

```
void (*fp)(void);
```

is a function pointer ‘fp’ returning void and having void as parameter, we 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];
```

We can write a constant function pointer as:

```
void (*const cfp) (void);
```

What about a constant function pointer having a constant int pointer as 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);
```

An array of three constant function pointers, having void as parameter and returning void. Can ‘fp’ allocated in ROM? Yes, because the array ‘fp’ itself is constant.

Here a little more complicated 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 more complicated here is the return value. Its actually a function pointer of a quite complex type. Here we do not explain where to put a const so that the destination of the returned pointer cannot be modified. Alternatively, the same function can be written much simpler 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;
}
```

Bye 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

When you are working in HLI assembler, you can define some ANSI C macros, containing HLI assembler statements.

Because HLI assembler is heavily Back End dependent, we are using here 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 the address of variable ‘var’ into Reg1
STORE Reg0,var ; Store Register 0 into variable ‘var’
```

Those HLI instructions are only used as a possible example, for real applications you have to replace the above pseudo HLI instructions with the real one of your target.

Defining a macro

A HLI assembler macro can be defined using the preprocessor directive ‘define’.

Example:

```
/* Following macro clears the register zero. */
#define ClearReg0 {asm CLR Reg0;}
```

In the source code, the macro is invoked in the following way:

```
ClearReg0;
```

the preprocessor will expand the macro in the following way:

```
{asm CLR Reg0;};
```

A 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 a HLI assembler block in a macro, but you can define a list of HLI assembler instructions.

Example:

```
/* Following macro clears the registers 0 and 1. */
#define ClearReg0And1 {asm CLR Reg0; asm CLR Reg1}
```

In the source code, the macro is invoked in the following way:

```
ClearReg0And1;
```

the preprocessor will expand the macro in the following way:

```
{asm CLR Reg0; asm CLR Reg1; };
```

You can define a HLI assembler macro on several lines using the line separator '\'.

Note:

This may enhance the readability of your source file, but the ANSI C preprocessor will still expand the macro on a single line.

Example:

```
/* Following macro clears the registers 0 and 1. */
#define ClearR0andR1 {asm CLR Reg0; \
                      asm CLR Reg1}
```

In the source code, the macro is invoked in the following way:

```
ClearR0andR1;
```

the preprocessor will expand the macro in the following way:

```
{asm CLR Reg0; asm CLR Reg1; };
```

Using Macro Parameters

A HLI assembler macro may have some parameters, which can be referenced in the macro code.

Example:

```
/* Following macro initializes the specified variable with 0.*/
#define Clear(var) {asm CLR var}
```

In the source code, the macro is invoked in the following way:

```
Clear(var1);
```

the preprocessor will expand the macro in the following way:

```
{asm CLR var1 ; };
```

Using Immediate Addressing Mode in HLI assembler Macros

When using 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 where the symbol # is specified should remain unchanged. This macro is valid for the rest of the file, where it is specified.

Example:

```
/* Following macro initializes the specified variable with 0.*/
#pragma NO_STRING_CONSTR
#define Clear(var) {asm LOAD #0,Reg0; asm STORE Reg0,var}
```

In the source code, the macro is invoked in the following way:

```
Clear(var1);
```

the preprocessor will expand the macro in the following way:

```
{ asm LOAD #0,Reg0; asm STORE Reg0,var1 };
```

Generating unique Labels in HLI Assembler Macros

When some labels are defined in a HLI Assembler Macros, if you invoke the same macro twice in the same function, the ANSI C preprocessor will generate twice the same label (once in each macro expansion). In order to generate unique labels you can use the special string concatenation operator of the ANSI C preprocessor (‘##’).

Example:

```
/* 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; }
```

In the source code, the macro is invoked in the following way:

```
copyMacro2(source2, destination2, 1);
copyMacro2(source2, destination3, 2);
```

During expansion of the first macro, the preprocessor will generate a label ‘lp1’ and during expansion of the second macro a label ‘lp2’ will be created.

Generating Assembler Include Files

In many projects it often make sense to use both a C compiler and an assembler. Both have different advantages, so the compiler uses portable and readable code and the assembler gives the user full control for time critical applications or for direct accessing the hardware.

Unfortunately neither the compiler can read include files of the assembler, nor the assembler can read the header files of the C 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 basic idea is that 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 is additionally 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.

Anywhere in the header file, the assembler output can be enabled with a pragma :

```
#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 general legal assembler constructs. Especially care must be taken for macros. The compiler does not check for legal assembler syntax when translating macros. Therefore macros containing elements not supported by the assembler should be in a section controlled by a ”#pragma CREATE_ASM_LISTING OFF”.

The compiler does only create a output file when the option -la is specified and the compiled sources do contain a #pragma CREATE_ASM_LISTING ON.

Simple 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 he generates the following file a.inc :

```
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
```

This file can now be included into any assembler file with the assembler INCLUDE directive. Then the content of the C variable Var_i can also be accessed from the assembler without any uncertain assumption about the alignment used by the compiler. Also whenever any C user adds a field to the structure Struct, then the assembler code must not be altered. Just the generated file a.inc must be generated again, which can easily be done with a make tool.

Usually the assembler include file is not created by every time the compiler reads the header file, but only in a separate pass when the header file has really changed. So the option -la is only specified when the compiler should generate a.inc. If the option -la is always present, the file a.inc is always generated and a make tool will always restart the assembler because the assembler files depend on a.inc. Such a makefile might be similar to this one :

```
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, but the order of EQU directive does matter for the assembler. If the assembler has problems with the 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 correctness of the define in any way.

The following example show some legal usages 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 and the octal 01234 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 § § / % & % / &  + * % ç 65467568756 86
```

Creates :

```
                        XREF i
AddressOfI              EQU  & i
ConstantInt             EQU  ( ( int ) 1 )
useUndef                EQU  UndefFkt * 6
Anything                EQU  § § / % & % / & + * % ç 65467568756 86
```

The AddressOfI macro does not assemble because the assembler does not know the C address operator &. Also other C specific operators as dereferenciation (*ptr) and others 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, so Mul7 does not occur in the listing. Also macros just defined by with no actual value as Nothing are not emitted.

The C preprocessor does not care about any syntactically content of the macro, but the assembler EQU directive does. The compiler has therefore no problems with the useUndef macro using the undefined object UndefFkt, but the assembler EQU directive requires that all used objects are defined.

The Anything macro finally shows that the compiler really does not care about the

content of a macro. The assembler, of course cannot treat this random characters.

All this sort of macros can be 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 the compiler will not emit anything for them.

Enums

Enums in C have a unique name and a defined value, so they are simply emitted by the compiler as EQU directive.

```
#pragma CREATE_ASM_LISTING ON
enum {
 E1=4,
 E2=47,
 E3=-1*7
};
```

Creates

```
E1                      EQU $4
E2                      EQU $2F
E3                      EQU $FFFFFFF9
```

Note that 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.

For all typedefs, the size of the newly defined type is specified. 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 structure offsets is build by using the typedef name and then the structure fields name after a 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
```

Note that even structure fields inside of structures are contained. In this case, the generated name contains all names of all whole path. If any element of the path does not have a name (e.g. an anonymous union), this element is not emitted.

For all bitfields members, the width and the offset is additionally emitted. The offset 0 specifies the least significant bit, which can be accessed with mask 0x1. The offset 2 the bit which can be accessed with mask 0x4. The width specifies the number of bits.

Notice that the offsets, bit widths and bit offsets given here are just examples. Different compiler may emit different values. In C the structure alignment and the bitfield allocation is up to the compiler, so only the actually used compiler can specify the correct values.

Functions

Functions declared are emitted by a XREF directive, so they can be used with the assembler.

Notice that function to be called from C but defined in assembler should not be emitted into the output file because the assembler does not allow the redefinition of labels declared with XREF. Therefore such function prototypes can be placed in an area started with a "#pragma CREATE_ASM_LISTING OFF".

```
#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 an XREF. Additionally for structures, the every field is defined with an EQU directive. For bitfields, the bit offset and bit size is additionally defined.

Variables in the SHORT segment are defined with a XREF.B to inform the assembler about the direct access.

```
#pragma CREATE_ASM_LISTING ON
struct A {
  char a;
  int i:2;
};
struct A VarA;
#pragma DATA_SEG SHORT ShortSeg
int VarInt;
```

Creates:

```
                        XREF VarA
VarA_a                  EQU VarA + $0
VarA_i                  EQU VarA + $1
VarA_i_BIT_WIDTH        EQU $2
VarA_i_BIT_OFFSET       EQU $0
                        XREF.B VarInt
```

The size of a variable is not explicitly written. To also emit the size, use a typedef with the variable type.

Notice that the offsets, bit widths and bit offsets given here are just examples. Different compiler may emit different values. In C the structure alignment and the bitfield allocation is up to the compiler, so only the actually used compiler can specify the correct values.

Comments

Comments inside of a region emitted with "#pragma CREATE_ASM_LISTING ON" are also written into the assembler include file on a single line.

Comments inside of a typedef, structure or variable declaration are either before or after the declaration written, but not inside of the declaration, if it contains out of multiple lines. So 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. A empty comment /* */ therefore inserts a 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
```

C++ Front End

C++

Overview

This chapter gives an overview about the implementation of the C++ language in this Compiler. It explains the main features of the language together with information about their implementation, because they may vary between different Compiler vendors.

One special chapter discusses the impacts of C++ for embedded applications.

Implementation

This implementation of C++ Compiler is mainly based on the book "Ellis/Stroustrup: The Annotated C++ Reference Manual".

This text cannot be a real introduction to the C++ language, programmers not familiar with C++ should read additional documentation. Beside the documents of the ANSI standardization group, we recommend the following books:

- "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

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 can be 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
- compactC++ library (iostream, new/delete, string, complex)
- Default Arguments
- Operators "new" and "delete"
- C++ libraries (streams, new/delete, strings, complex, bitset, STL)

Following C++ features are NOT SUPPORTED YET:

- Static local class objects
- Non-constant initialization of global non-class objects
- Destructor call of global 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
- C++ libraries "exception" and "valarray"
- Full C++ library (bits, dynarray, exception, STL)

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

C and C++ sources can be mixed. Following points are important:

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. With this macro it is possible to distinguish on source level between ANSI-C and C++ Compilers or to make sources compatible 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 usage of the macro in ANSI-C header files to make them compatible booth for ANSI-C and C++.

```
#ifdef __cplusplus
  extern "C" { /* C++: use C calling convention */
#endif

void foo(int, double);
/* ... */

#ifdef __cplusplus
  }
#endif
```

Differences between ANSI-C and C++

Empty Parameter Declarations

A function like f() has no parameters in C++, but in C it has any number:

```
void foo();
  /* C++   : same as 'void foo(void)' */
  /* ANSI-C: undefined number of parameters  */
```

Return Values

Functions returning a value must have a correct return-statement in C++. In C they don't have to.

Example:

```
int f(void) {
  int i;
  i=3;
} /* C legal, C++ error: return required. */
```

Implicit Parameter Declaration

Functions must be declared before using.

Example:

```
void f(void) {
  g(); /* ANSI-C legal, C++ error: missing prototype */
}
```

See also Option -Wpd

Constant Variables

Constant variables must be initialized in C++. In C they don't have to.

```
#ifdef __cplusplus
  const int i = 3; /* C++ requires initialisation */
#else
  const int i;  /* legal for ANSI-C */
#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, because the Linker cannot allocate an object with size zero. Non-empty classes derived from empty classes does not contain such a dummy character.

Example:

```
class EmptyClass {} ;  // sizeof(EmptyClass) returns 1!
class DerivedFromEmtpy : EmptyClass {
  int a;
};

int i, j;

void main(void) {
  i = sizeof(EmptyClass);       /* == 1 */
  j = sizeof(DerivedFromEmtpy); /* == 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, if the programmer knows, how to avoid the Compiler to produce inefficient code, it’s possible to get small code. This chapter provides you with a few hints, how to write small C++ programs suitable to the embedded system world. Depending on your need for efficient code, you can choose between the languages Full ANSI Draft C++, Embedded C++ (EC++) and compactC++ (cC++) with the option -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. So you can select, which features have to be switched off/on depending on your needs.

About Compiler Generated Functions

The Compiler generates special member functions for classes, if necessary. This increases the code size of a module very much. However, often those generated functions are not called in the application. So the Linker will throw them away anyway. But alternatively you can avoid the Compiler to generate those functions by using the option -Cn=Ctr.

If the 'Default Copy Constructor' should not be generated automatically, there are two different solutions:

Use the compiler option -Cn=Ctr to do not create compiler defined functions in C++ or generate it by hand:

```
class A {}

A::A(){}         // is the constructor
A::A(const A&){} // is the copy constructor
A::~A(){}        // is the destructor
```

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: An inlined function member is implicitly set as STATIC function member.

Another way is to use the option -Oi.

Use Class Templates

You can parametrize classes with templates. So you can avoid class Constructors to

perform unnecessary tasks at runtime.

Example:

```
/*******************************************/
/* 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 much code. Each instantiation of a template function generates code for the whole function with those 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

Using virtual member functions, virtual function tables are generated. Those tables contain a function pointer and an associated delta value used to access the base class. Usually those delta values do have a (signed) 16bit type, allowing class

objects with a size of 32kByte each. But to save ROM usage for such tables, the type of this delta value can be changed using the -Tvtd option. E.g. if all class objects are smaller or equal 127 bytes, then the virtual table delta type can be changed to a 1 bytes (signed!) type using -Tvtd. If there should be a class object not fitting into this type, an error message (C1393) will be generated.

Avoid Pointer to Member Parameters and Pointer to member Returns

Cause the pointer to member type is not a pointer (see “The annotated C++ reference manual” from ELLIS-STROUSTRUP), the compiler has to handle a data structure. Thus the following example

```
class A{
public:
  void fct(void){ /*........*/ }
};
void f(void (A::*pm)(void)){
  A a;
  (a.*pm)();
}
void main(void){
  f(&A::fct);
}
```

will produce 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. In case of pointer to member as parameter or as return, no information are available at the function call statement, that is why we have to handle those tests at runtime.

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. Such copy constructor calls are not efficient. Better pass references instead.

Example:

```
/*********************************************/
/* 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. But they are very inefficient in use. Thus try to avoid them.

Avoid Virtual Base Classes

Virtual base classes are not efficient to use. So try to avoid them. See also 'Short C++ tutorial'

Avoid NULL-Check of Class Pointers

Operations with pointers to related classes always need separate NULL-checks before adding offsets from base classes to inherited classes. So try to 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 half way down the function, you should create it half way down the function when it can be initialized correctly. Don't initialize it to an "empty" value at the top then "assign" it later. The reason for this is runtime performance.

Examples:

```
//bad:
void f() {
  int i=0;
  /* ... */
  i=5;
  /* using i */
}
```

//good:

```
void f() {
  /* ... */
  int i=5;
  /* using i */
}
```

About Code size

Many people are surprised by how big the object files are, especially if the source code is trivial. A simple "hello world" program can generate an executable that is larger than most people expect. The reason is following: For example, 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 throws away not used functions and objects. So the size of the executable will decrease in comparison to the object file.

C++ Overhead

The following table give an overview about the possible overhead (ROM/RAM) using a C++ feature compared with standard ANSI-C:

C++ Feature	Overhead
C++ comments (e.g. ‘//’)	none.

C++ Feature	Overhead
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)	huge, because 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	huge 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

1. Classes

The C++ class concept is a generalization of the C notion of a structure, or - looking the other way - the C concept of a structure is a simple variant of the C++ class concept. In particular, 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. Structure's 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 can be 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 can override 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 can be expressed using virtual base classes.

Classes can be declared abstract to ensure that 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 may be declared “static” in the class declaration. There is only one copy of a static data member, 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 can be assigned to base class objects. So the base class part of the derived class object is copied to the base class object.

Pointers to class objects can be assigned to pointers to base class objects. So 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 can be private, that is, its name can be used only by member functions, member initializers, and friends of the class in which it is declared.

Protected

That is, its name can be 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

That is, its name can be used by any function or initializer.

Example (public, private, protected)

```
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

Those access specifiers can also be applied to base classes. If the base class is private, 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.

Example (public, private, protected base class)

```
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.

Example:

```
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.

Following tasks are performed:

- Assign vtptr (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

Notes:

- Constructors can be overloaded (the one without parameters is called “default constructor”).
- Global class objects call their constructors at the time all global objects get initialized.

Example:

```
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 destructor of a class is called before an object of that class is destroyed.

Following tasks are performed:

- Execute function body of destructor

- Call destructors of base classes and member class objects (in reverse declaration order)

Notes:

- Destructors can not be overloaded.

- Global class objects call their destructors at the end of the program (after the main() routine finished)

```
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, and 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.

4. Virtual Functions

A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Example:

```
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();  // call A::f
```

```
    pta = &b;
    pta->f();  // call 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 can be 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 can be suppressed by explicitly qualifying the function name using the scope-resolution operator '::'.

Example:

```
  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 do not affect the access control for a given type.

Example:

```
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 can be 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 can be 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

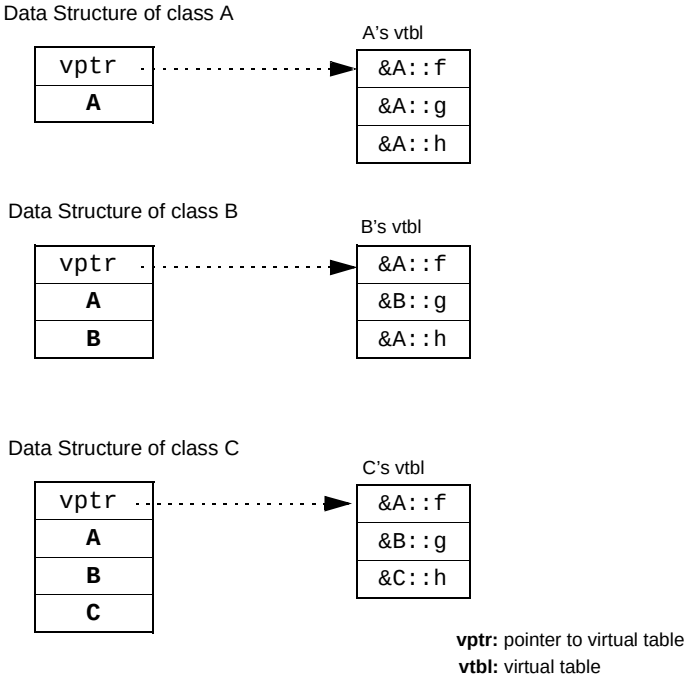
The implementation of virtual-functions mechanism is based on the following concept. Virtual functions can be implemented with a table of pointers to virtual functions, the vtbl. A pointer object is introduced in the class scope, the vptr. It points to the vtbl.

Single Inheritance and Virtual Functions

This is the objects representation of the following declarations:

```
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);
```

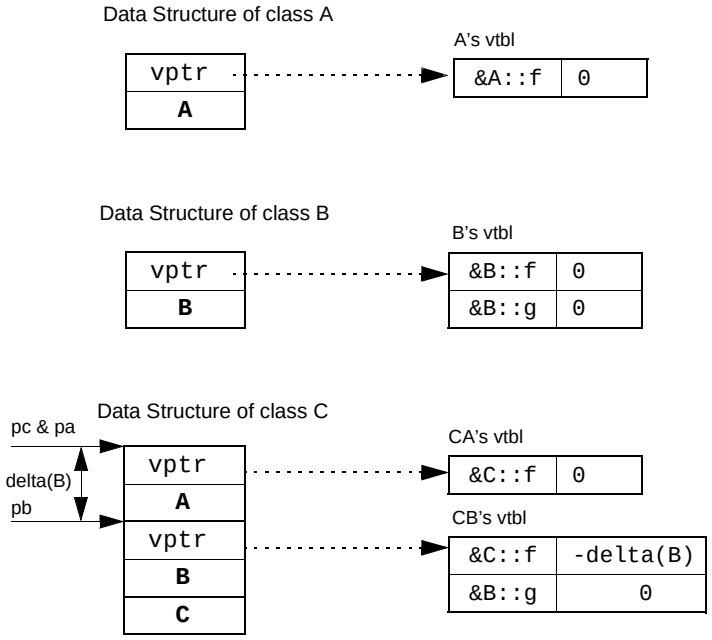
becomes to something like:

```
(*(pc->vptr[1]))(pc,2);
```

Multiple Inheritance and Virtual Functions

This the objects representation of the following declarations:

```
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 can be 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 can be 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
```

HI-CROSS+ Linkers support 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 can be 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 it can be an indirect base class more than once.

Example:

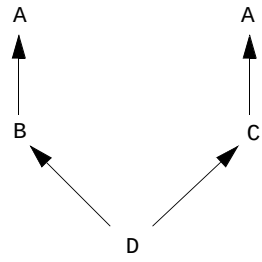
```
class A{
public:
  // member list
};
class B : public A, public A{  //Ill-formed
public:
  // member list
};
class A{
public:
  int next;
  // member list
};
class B : public A {
public:
  // member list
};
class C : public A {
public:
  // member list
};
class D : public B, public C {  //Well-formed
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 can be used to specify which sub-object is meant. The body of the function

```
C::f() { A::next = B::next; } // Well-formed
```

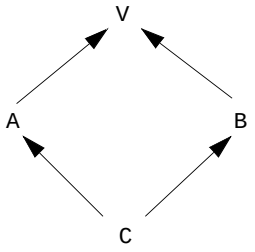
Without the A:: or B:: qualifiers, the definition of C::f above would be ill-formed because of ambiguity.

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.

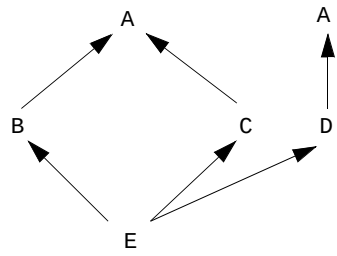


For another example:

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

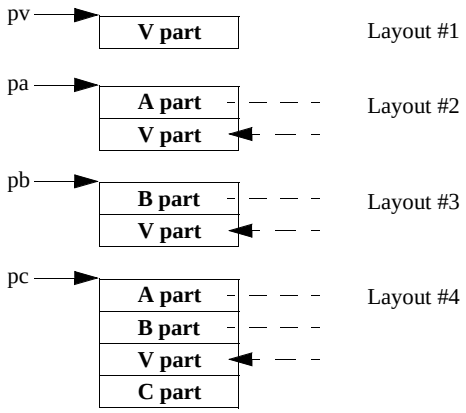
Suppose we have the following inheritance hierarchy.

```
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 following figure:

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 VBPTR
  for each base class
    if current base class contains a VBPTR
      set StandAlone to 0
…
for each constructor
  if constructor of class containing a VBPTR
    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 these declarations,

```
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 following figure:

Virtual Base Classes and Casting

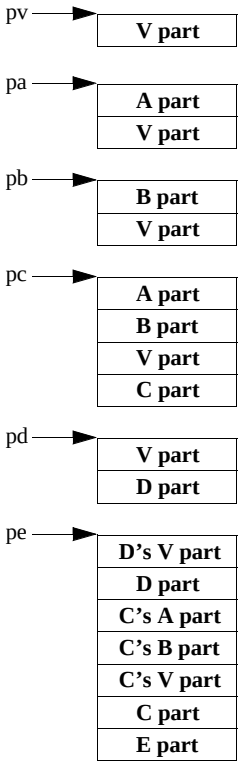
One can cast from a derived class to a virtual base class. The generated code will use the pointer to the virtual base class stored in the derived class.

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 can be reached through more than one path. Because different access control can be 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 can be 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. Every template argument must be used in the argument types of a function template.

Example: Several sorting functions for different types should be 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 2 kinds of template arguments: type specification and expression specification.

Example:

```
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.

Example:

```
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 for each template of a given name in a program. There can be many declarations. The definition is used to generate 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 using. Template functions don't 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.

Example (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 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(); //generate a linking error cause no object is defined.
  b.f();    //call the function defined in B class.
}
```

Pure virtual function are used to make the class containing an abstract class. Abstract class mechanism supports the notion of a general concept, of which only more concrete variants can actually be used. An abstract class can also be used to define 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 supports the notion of a general concept, such as a shape, of which only more concrete variants, such as circle and square, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations.

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. 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 in the class declaration. A pure virtual function need be defined only is explicitly called 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,

```
class B : public A{
  public:
    void f(void){}
    void g(void);   // must be define somewhere
};
```

would make class B non-abstract and a definition of B::g() must be provided somewhere.

Member 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.

```
class A {
  public:
    virtual void f(void) = 0;
    A(){
      f();  // ERROR
    }
};
```

Note that a pure virtual can be defined. It can be 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

Example 1:

```
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)

Example 2:

```
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

Example 3:

```
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)

Example 4:

```
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

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 will be 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 can be 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 can be 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 an array of class, the default constructor of each element is called. Therefore the compiler generates a loop going through all array elements.

Example:

```
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 can be defined globally or within a class. The global declaration is already done internally in the compiler. The member operator "new" is a static member.

Example:

```
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 an array of class 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 can be defined globally or within a class.

Example:

```
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 can be defined globally or within a class. The global declaration is already done internally in the compiler. The member operator "delete" is a static member.

Example:

```
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. Thus a program calling new and delete needs to be linked with the appropriate library, otherwise those operators must be implemented by the programmer himself. If you don't provide any implementation of "new" and "delete" to your program, the compiler won't complain, because there are internal declarations of the global operators. But the linker will complain, if it doesn't find any implementation.

10. Pointer to Members

As initially defined, the C++ language provided no way of expressing the concept of a pointer to a member of a class. When a pointer to a member function was needed, in an error handling function by example, users had to subvert the language’s type checking.

The type of a nonmember function,

```
int fct(char *),
```

for example, is

```
int (char *),
```

and a pointer to such a function is of type

```
int (*) (char *).
```

Pointers to nonmember functions can be declared and used like this:

```
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 function fctMbr and object mbr are declared, and an object obj of that class, 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 pointer to member of class A is A::* and the notation for taking the address of a member of a class A is &A::. Thus one 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 doesn't 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 *). One can now write

```
//declare and initialize pointer to member function
int (A::*pmf)(char *) = &A::f;
A obj;
//call function through pointer for the object obj
int j = (obj.*pmf)(“hello”);
```

The syntax is not the most readable one can imagine. It is, however, consistent with the C declarator syntax.

A pointer to member function can be 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 can be called through a pointer to an object. Consider the following code:

```
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() will be invoked. Naturally, the implementation looks in c’s table of virtual functions exactly as it would for a call to a virtual function identified by name rather than by a pointer.

Pointers to Static Members

Pointers to static members are ordinary pointers. Thus 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 nonmember objects. Similarly, a pointer that points to a static member is dereferenced as an ordinary pointer.

C++ Name Encoding, Type-safe Linkage

One of the advantages using C++ is to have saver type linkage than in ANSI-C.

In ANSI-C, linkage is done by name, thus 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/declared with 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' */
}
```

C++ object files have different name encoding of the objects than ANSI-C object files. Thus, if you link C++ object files with ANSI-C object files, the imported functions must have the linkage of the language, where they are exported. See 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

Modifier	Encoding
signed (only for characters!)	S

Types	Notation	Encoding
simple	Complex	7Complex
qualified	X::YY	Q21X2YY
pointer	*	P
reference	&	R
array	[10]	A10_
function	()	F
global operator	op ()	FG
pointer to member	S::*	M1S

Operator	Encoding
*	__ml
/	__dv
%	__md
+	__pl
-	__mi
>>	__rs
<<	__ls
==	__eq
!=	__ne
<	__lt
>	__gt
<=	__le
>=	__ge

Operator	Encoding
&	__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
() (function call)	__cl
[] (subscripting)	__vc
constructor	__ct
destructor	__dt

Operator	Encoding
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 can be handled directly by the Compiler, and with a little help from the programmer it is possible to reach denser code. Some Compiler options or usage of SHORT segments (if available) can help you to generate compact code with the Compiler.

Compiler Options

Usage of following compiler options will help 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 will prevent the compiler to reload the address of the pointer for each access. An index register may hold the pointer value over statements if possible.

NOTE: This option may not be available for all targets.

-Cni: Non integral promotion on integer

When this option is specified, the ANSI C integral promotion does not apply to character comparison or arithmetic. This can save a big amount of code.

-Oi: Inline Functions

For C++ functions, use the inline keyword or the command line option -Oi. Defining a function before it is used helps the Compiler to inline it:

```
/* ok */
void foo(void);
void main(void) {
  foo();
}
void foo(void) {
 // ...
}
```

```
/* better! */
void foo(void) {
  // ...
}
void main(void) {
  foo();
}
```

This also helps the compiler to use a relative branch instruction instead an absolute one.

-Oc: Common Subexpression Elimination

The compiler tries to perform CSE: instead evaluating the same expression several times, the Compiler evaluates it once and stores it on the stack for later retrieval.

NOTE: This option may not be available for all targets.

SHORT Segments

Variables allocated on the direct page (between 0 and 0xFF) can be accessed using the direct addressing mode. You can tell the Compiler that some variable are allocated on the direct page, if you define them in a SHORT segment (not available for all targets).

Example:

```
#pragma DATA_SEG SHORT 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 segment, the external declaration for these variables must also specify that they are allocated in a SHORT segment. The External definition of the variable defined above will look like:

```
#pragma DATA_SEG SHORT myShortSegment
extern unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
extern unsigned int myvar3, myVar4.
```

Then 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;
    _DATA_ZEROPAGE, myShortSegment INTO  Z_RAM;
```

```
END
STACKSIZE 0x60
VECTOR 0 _Startup /* set reset vector on _Startup */
```

Be careful: The linker is case sensitive. The segment name must be identical in the C and PRM file.

Defining IO Registers

IO Registers are usually base at address 0. In order to tell the compiler it should use direct addressing mode to access the IO registers, these registers can be defined in a SHORT section (if available) based at the specified address.

In the C source file, the IO register are defined 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 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;
    _DATA_ZEROPAGE                 INTO  Z_RAM;
    SCIRegs                        INTO  SCI_RG;
END
STACKSIZE 0x60
VECTOR 0 _Startup /* set reset vector on _Startup */
```

Be careful: The linker is case sensitive. The segment name must be identical in the C and PRM file.

Programming Guide Lines

Following few programming guidelines can also reduce the code size. Many things are optimized by the Compiler, but if the programming style is very complex or if it forces the Compiler to perform special code sequences, code efficiency is maybe not what you expect.

HLI Assembly

If possible, do not mix High Level Inline (HLI) Assembly. Using HLI assembly may affect the register trace of the compiler, because 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, e.g. it cannot hold variables into registers over HLI statements. Normally it is better to place special HLI code sequences into separate functions, but then there is the drawback of an additional call/return. But placing HLI instructions into separate functions (and module) will simplify porting the software to another target too.

Example (not recommended):

```
void foo(void) {
  /* some local variable declarations */
  /* some C/C++ statements */
  asm {
    /* some HLI statments */
  }
  /* maybe other C/C++ statements */
}
```

Example (recommended):

```
/* hardware.c */
void special_hli(void) {
  asm {
    /* some HLI statments */
  }
}

/* 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 program complex may also result into complex code. In general it is the job of the compiler to optimize also very complex functions, but some rules may help the compiler to generate efficient code.

If your target does not support powerful postincrement/postdecrement/preincrement/predecrement instructions, it is not recommended to use the '++' and '--' operator in complex expressions. Especially postinc/dec may result additional code:

```
a[i++] = b[--j];
```

Better write above statement as

```
i--; a[i] = b[j]; j--;
```

Using it in simple expressions as

```
i++;
```

is not a problem, of course.

Additionally, avoid assignments in parameter passing or side effects (as '++' and '--'). The problem is that 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:

```
i = 3;
foo(i++, --i);
```

In the above example, foo() will be 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 (!= 0) is TRUE. Instead using an 'int' (usually 16 or 32 bit), it may be better to use an 8bit 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

The printf/scanf in the ANSI library can be reduced e.g. if no floating point support (%f) is used. See ANSI library reference and the printf.c/scanf.c in your library for details how to save a lot of code (not using float/doubles in printf may result in half the code!).

Bitfields

Using bitfields to save memory may be a bad idea, because often bitfields has to produce a lot of additional code. For ANSI-C compliance, bitfields do 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 returning a struct. Except if the struct can be returned in a single register, returning a struct may force the Compiler to produce a lot of 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, e.g. avoiding some copy operations, but in general it is quite more complex than it looks like. Returning a struct by value may use a lot of execution time, 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 indirect into the destination (4). This approach reduces stack usage, avoids to copy structs and results in denser code. Note that the Compiler may also the above sequence (if supported), but for rare cases the above sequence is not 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 global ones results into better manageability of the application, because side effects are reduced or totally avoided. Using local variables/parameters reduces global memory usage, but increases stack usage. Thus stack access capabilities of the target influences the code quality. Depending on the target capabilities access to local variables may be very inefficient. Reasons are 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). Additionally 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, but this deteriorates maintainability and also may waste global address space. As a solution the Compiler may offer an option to overlap parameter/local variables using a technique called ‘overlapping’: Local variables/parameters are allocated as global ones, and the linker overlaps them depending on their usage. For targets with limited stack (e.g. no stack addressing capabilities) this often is the only solution, but with this solution the code is not reentrant (no recursion is allowed).

Parameter Passing

Avoid parameters which exceeds the data passed through registers (see Back End).

Unsigned Data Types

Using unsigned data types is normally a ok because signed operations are normally much more complex than unsigned ones (e.g. shifts, divisions and bitfield operations). But normally 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 absl()

Instead calling abs() and absl() in the stdlib, it is better to use the corresponding macro M_ABS defined in stdlib.h:

```
/* 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: because M_ABS() is a macro,

```
i = M_ABS(j++);
```

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 handle also 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 does not return the 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);
}
```

But 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 function does not return the dest and assumes count > 0 */
  do {
    *((uchar *)dest)++ = *((uchar*)source)++;
  } while(count--);
}
```

Thus replacing calls to memcpy() with calls to memcpy2() may save a lot of 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 then ‘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 segment using

```
#pragma DATA_SEG SHORT MySeg
```

Qualifiers

Use whenever possible the ‘const’ qualifier to help the compiler. ‘const’ objects are placed into ROM for the HIWARE object file format, if the command line option -Cc is given

Motorola HC08 Back End

The Back End is the target–dependent part of a Compiler, containing mainly the code generator. This section discusses the technical details of the Back End for the M68HC08 family.

Memory Models

SMALL Model

The SMALL memory model is the default. All pointers and functions are assumed to have 16 bit addresses, if not explicitly specified differently. In the SMALL memory model code and data must be in the 64k address space. The SMALL memory model can be chosen with the option -Ms.

BANKED Model

The difference of the BANKED to the SMALL memory model is that in the BANKED memory model functions do have the far calling convention by default. All pointers, including function pointers are 16 bit wide. In the banked memory model data must be in a not banked area of the 64k standard address space.

Note because only 16 bit function pointer are available, function pointer can only point to functions explicitly marked as near. See the chapter far and near calling convention for details about the far calling convention.

The BANKED memory model can be chosen with the option -Mb.

Non–banked Memory

Some parts of an application always must be in non–banked memory, in particular:

- the prestart code (segment `_PRESTART`)
- the startup code (segment `NON_BANKED`) and the startup descriptor(s) (segment `STARTUP`)
- all functions used as function pointer target in the ANSI library (segment `NON_BANKED`)
- all runtime support routines (segment _RUNTIME)
- all variables, strings, constants and copy down (segment COPY). Banking available for code only.

- all interrupt handlers, because trap vectors are only 16 bits wide.

Example of a Linker Parameter File (extract):

```
SECTIONS
    MY_NEAR_ROM = READ_ONLY  0x1000 TO 0x7FFF;
PLACEMENT
    _PRESTART, NON_BANKED, _RUNTIME, COPY, STRINS
       INTO MY_NEAR_ROM;
```

For more information on these segments, see the Linker manual.

Non-ANSI Keywords

Following table gives an overview about the supported non-ANSI keywords:

Keyword	Supported for		
	data pointer	function pointer	function
far	no	no	yes
near	no	no	yes
rom	no	no	no
uni	no	no	no
interrupt	no	no	yes

The keywords far and near are only applicable to function.

Data Types

This section describes how the basic types of ANSI-C are implemented by the MC68HC08 Back End.

Scalar Types

All basic types may be changed with the -T option (only available with HI-CROSS+). Note that all scalar types (except char) are without signed/unsigned qualifier by default signed, e.g. ‘int’ is the same as ‘signed int’.

The sizes of the simple types are given by table below together with the possible formats using the -T option (only available with HI-CROSS+):

Type	Default format	Default value range		Formats available with option -T
		min	max	
char (unsigned)	8bit	-128	127	8bit, 16bit, 32bit
signed char	8bit	-128	127	8bit, 16bit, 32bit
unsigned char	8bit	0	255	8bit, 16bit, 32bit
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 unsigned! This default can be changed with the -T option (only available with HI-CROSS+).

Floating Point Types

The Compiler supports the two IEEE standard formats (32 and 64 bits wide) for floating point types.

By default, type `float` is implemented in IEEE 32bit format, while `double` is in IEEE 64bit format. If you care more about speed than accuracy of `double` arithmetic, you may give command line option -Fd. If this option is given, both `float` and `double` are implemented using the IEEE 32bit format.

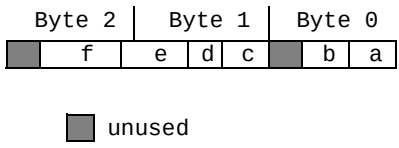
The option -T may be used to change the default format of float/double.

Type	Default format	Default value range		Formats available with -T
		min	max	
float	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64
double	IEEE64	2.2259738585972014E-308	1.7976931348623157E+308	IEEE32, IEEE64
long double	IEEE64	2.2259738585972014E-308	1.7976931348623157E+308	IEEE32, IEEE64
long long double	IEEE64	2.2259738585972014E-308	1.7976931348623157E+308	IEEE32, IEEE64

Bit Fields

The maximum width of bit fields is 16 bits; they never cross a word boundary. The allocation unit is a byte, the Compiler uses words only if a bit field is wider than 8 bits or using bytes would cause more than 2 bits to be left unused. Allocation order is from the least significant bit up to the most significant bit in the order of declaration. The following figure illustrates this allocation scheme.

```
struct {
  unsigned int a:3;
  unsigned int b:3;
  unsigned int c:3;
  unsigned int d:2;
  unsigned int e:4;
  unsigned int f:5;
} S;
```



The following example demonstrates a simple C Source with various ways to access bitfields together with the produced code:

```
/* demonstration of bit field instruction usage of HIWARE HC08
Compiler */
#pragma DATA_SEG SHORT _zpage /* place following variables into zero
page */
#define BIT_SET(x,bitNo) ((x) |= 1<<(bitNo))
```

```
#define BIT_CLR(x,bitNo) ((x) &= ~(1<<(bitNo)))
char i, j;
struct {
  unsigned int b0:1;
  unsigned int b1:1;
} B;
void main(void) {
  /* demo using BSET/BCLR/BRSET/BRCLR */
  if (i&1) { /* BRCLR */
    /* BCLR: clearing a bit */
    i &= ~1;        /* using normal ANSI-C */
    BIT_CLR(j,0);   /* using a macro */
    asm BCLR 0,i    /* using inline assembly */
  }
  if ((i&1) == 0) { /* BRSET */
    /* BSET: setting a bit */
    i |= 1;         /* using normal ANSI-C */
    BIT_SET(j,0);   /* using a macro */
    asm BCLR 0,j    /* using inline assembly */
  }
  if (i&4) { /* BRCLR */
    i &= ~4; /* BCLR */
  }
  if ((i&4) == 4) { /* BRSET */
    i |= 4; /* BSET */
  }
  /* demo using bitfields in ANSI-C (warning: not portable, depends
on compiler
     how bitfields are allocated! */
  if (B.b0) {
    B.b1 = 1;
  } else if (B.b1) {
    B.b0 = 0;
  }
}
#if 0
   13:  void main(void) {
   14:    /* demo using BSET/BCLR/BRSET/BRCLR */
   15:    if (i&1) { /* BRCLR */
   16:      /* BCLR: clearing a bit */
00000000 010006           BRCLR  0,i,*6       /abs = 0009
   17:      i &= ~1;        /* using normal ANSI-C */
00000003 1100             BCLR   0,i
   18:      BIT_CLR(j,0);   /* using a macro */
00000005 1100             BCLR   0,j
   19:      asm BCLR 0,i    /* using inline assembly */
   20:    }
00000007 1100             BCLR   0,i
   21:    if ((i&1) == 0) { /* BRSET */
```

```
   22:      /* BSET: setting a bit */
00000009 000006           BRSET  0,i,*6       /abs = 0012
   23:      i |= 1;         /* using normal ANSI-C */
0000000C 1000             BSET   0,i
   24:      BIT_SET(j,0);   /* using a macro */
0000000E 1000             BSET   0,j
   25:      asm BCLR 0,j    /* using inline assembly */
   26:    }
00000010 1100             BCLR   0,j
   27:    if (i&4) { /* BRCLR */
00000012 050002           BRCLR  2,i,*2       /abs = 0017
   28:      i &= ~4; /* BCLR */
   29:    }
00000015 1500             BCLR   2,i
   30:    if ((i&4) == 4) { /* BRSET */
00000017 050002           BRCLR  2,i,*2       /abs = 001C
   31:      i |= 4; /* BSET */
   32:    }
   33:    /* demo using bitfields in ANSI-C (warning: not portable,
depends on compiler
   34:       how bitfields are allocated! */
0000001A 1400             BSET   2,i
   35:    if (B.b0) {
0000001C 010003           BRCLR  0,B,*3       /abs = 0022
   36:      B.b1 = 1;
0000001F 1200             BSET   1,B
00000021 81               RTS
   37:    } else if (B.b1) {
00000022 0300FC           BRCLR  1,B,*-4       /abs = 0021
   38:      B.b0 = 0;
   39:    }
00000025 1100             BCLR   0,B
   40:  }
00000027 81               RTS
#endif
```

Pointer Types, Function Pointers

Pointer variables are always two bytes wide. This also includes all function pointers.

Division and Modulus

The results of the "/" and "%" operators also are not properly defined for signed arithmetic unless both operands are positive. The implementation is best explained with the following example:

```
 22 / 7 ==  3;   22 %  7 == +1
 22 /-7 == -3;   22 % -7 == +1
-22 / 7 == -3;  -22 %  7 == -1
-22 /-7 ==  3;  -22 % -7 == -1
```

Structured Types, Alignment

Objects allocated in memory are *not* aligned; local variables are generally allocated on the stack (which is growing downwards) unless command line option –Cv is given, in which case some of these local variables may be allocated as global variables in segment `_OVERLAP`.

Like objects, elements of an array and `struct` or `union` members are not aligned.

The most significant part of a simple variable always is stored at the low memory address (big endian).

Note: The Compiler is free to align variables and fields. You should always use implementation-independent access.

Object Size

The maximum size of an object is 32 kBytes.

Register Usage

The Compiler uses all the standard registers of the MC68HC08; The stack pointer `SP` is used as stack pointer (of course), but also as frame pointer.

Call Protocol and Calling Conventions

Argument Passing

The C calling convention is used for all functions. The caller pushes the arguments from right to left. After the call the caller removes the parameters from the stack again.

If a function with a fixed number of arguments has arguments with a size of one byte (`char`), the first such argument from the left is *not* pushed but passed in the `A` register. This results in shorter code because pushing this parameter can be saved.

Return Values

Function results usually are returned in registers, except if the function returns an object with a size greater than 2 bytes. Depending on the return type different registers are used:

Return Type	Registers
`char` (signed or unsigned)	`A`
`int` (signed or unsigned)	`X:A`
pointers/arrays	`H:X`
function pointers	`H:X`

Returning Large Objects

Functions returning objects that are larger than 2 bytes are called with an additional parameter, which is passed in `H:X`. This parameter is the address where the object should get copied to.

Far and Near Calling Convention

The HC08 compiler supports two different calling conventions.

Near Calling Convention

Calls with the near calling convention are done with built-in machine instruction JSR. This instruction pushes the own return address (16 bit) on the stack and then continues the execution at the target. For returning to the called, the return address must be reloaded with a RTS.

The near calling convention is the default for the SMALL memory model.

Here an example, mainly for comparison with the far calling convention :

```
void near f() {
}
void g(void) {
  ...
  f();
  ...
}
```

Results in the following code :

The callee f:

```
...
RTS
```

The caller f:

```
...
JSR f
...
```

Far Calling Convention

The far calling convention is not directly supported by the hardware. Therefore to make a far call, a runtime routine is used. For absolute far calls the address of the target is encoded into the code directly behind the call to the runtime routine.

The stack contains two return addresses. First the return address of the caller as 24 bit address and then the return address of the runtime routine as 16 bit address. Therefore all parameters on the stack must be accesses with an offset enlarged by three compared with the near calling convention.

Except for the offsets to stack objects, the argument passing is identical for the near and far calling convention except the stack . From the callers point of view just a runtime routine is called instead of the target address. The target address is encoded as 24 bit address in the code directly behind an absolute call to the runtime routine.

The far calling convention is the default for the BANKED memory model.

Here the example for the far calling convention :

```
void far f() {
}
void g(void) {
  ...
  f();
  ...
}
```

Results in the following code :

The callee f:

```
...
RTS ; same as near calling convention !
```

The caller f:

```
...
```

```
JSR _FAR_CALL
DCB f:PAGE
DCW f
...
```

Specifying the Calling Convention

The calling convention can be specified with two different ways :

- with the pragma CODE_SEG modifiers NEAR and FAR
- with the keywords near and far in front of the function

If none of the two possibilities is used, the calling convention depends on the memory model. The default is near for the SMALL memory model and far for the BANKED memory model.

The first possibility has the advantage that also a segment name can be specified. This way far functions can be put into banked segments and near into non banked segments. With the keyword possibility the allocation of near and far functions must be done separately.

Example

```
void near f1_near(void); // near function;
void far f2_far(void);  // far function
#pragma CODE_SEG NEAR MY_NEAR_SEG
void f3_near(void);  // near function in segment MY_NEAR_SEG
#pragma CODE_SEG FAR MY_FAR_SEG
void f4_near(void);  // far function in segment MY_FAR_SEG
#pragma CODE_SEG DEFAULT
void f5(void);  // near in SMALL, far in BANKED memory model
```

Far call runtime routine

This runtime routine extends the return address of the caller to 24 with the actual value of the page register. Then it pushes it’s own return address, sets the page address of the target and finally calls the target. When the target is finished, it returns with an RTS to the runtime routine. Then this runtime routine restores the old page address and returns to the caller. To achieve maximal code density the runtime routine preserves all registers except the condition code.

The runtime routine for absolute calls is called _FAR_CALL and its use is explained above.

Adaptation of the far calling runtime routines to the Hardware

The runtime routines for far calls uses the define “pagereg” in the file “farcal08.c”

to access the page register. This define is at the start of the module “farcal08.c” and must be adapted to the actual page register of the hardware, if it has not already the right value. When page address “pagereg” has been changed, the “farcal08.c” module must be compiled and explicitly linked as object file prior to the ANSI library to the application.

Note: A module farcal08.o is contained in the ANSI library. The own object file must be used instead of this. This can be obtained by the order in the link parameter file.

Stack usage during a far call

Although the HC08 has a 16 bit stack, the stack usage should be as small as possible. The following parts are on the stack during a call instead of the 16 bit return address for the near calling convention:

- Return address of the Caller (16 bit)
- Page of the Caller (8 bit)
- Return address of the runtime routine

Inside of the runtime routine additional stack space is used, but it is released before the target function is called.

Function Pointer

The current implementation does only support absolute far calls. Function pointer can only point to functions using the near calling convention.

This restriction has especially consequences in the banked memory model. Because there function are using by default the far calling convention, only function explicitly marked as near can be used as call target.

Stack Frames

Frame Pointer

Functions normally have a stack frame containing all their local data. The Compiler does not set up an explicit frame pointer, but local data and parameters on the stack are accessed relative to the SP register.

Entry Code

Normal *entry code* is a sequence of instructions reserving space for local variables:

```
PSHX      ; Only for return of large object
PSHH      ; (dest.address of the object)
PSHA      ; Only if there's a register parameter
AIS #(-s) ; Reserve space for local variables
```

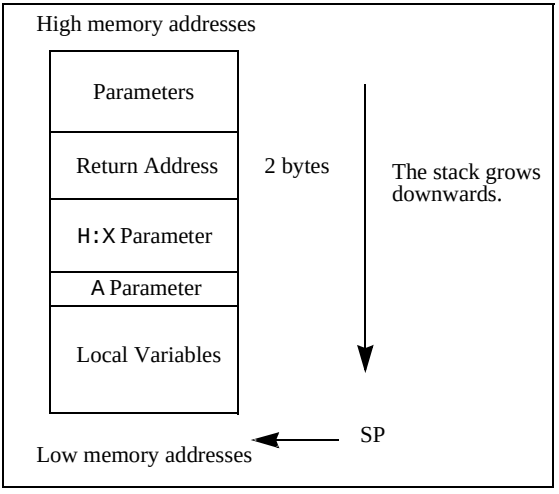
where `s` is the size (in bytes) of the local data of the function. There is no "static link", and the "dynamic link" is not stored explicitly.

Exit Code

Exit code has to remove local variables from the stack and return to the caller:

```
AIS #(t) ; Remove local stack space,
         ; including an eventual register parameter
PULH     ; Only for return of large object
PULX     ; Only for return of large object
RTS      ; Return to caller
```

If the function returns a large object, the `H:X` register always contains the objects's address after the return. The Figure below shows the stack frame of a function with a register parameter, returning a large object.



Pragma TRAP_PROC

This pragma defines an interrupt routine, i.e. a function with this pragma active is terminated by an `RTI` instruction instead of an `RTS`. Also, the `H` register is saved and restored at entry resp. exit of the interrupt routine.

Since runtime routines use a total of 7 bytes global memory for temporary variables

(in segment _DATA_ZEROPAGE, see Linker), these variables generally have to be saved and restored, too. However, if the interrupt routine doesn't call any other procedure, this saving and restoring is omitted.

These global variables are also *not* saved if you use the second form of this pragma:

```
#pragma TRAP_PROC SAVE_NO_VARS
```

In this case, above temporary variables of the runtime support routines are not saved. This saves time, but implies that the interrupt routine does not call any other function including runtime support routines.

Interrupt Vector Table Allocation

The Compiler provides a non-ANSI compliant way to specify directly the interrupt vector number in the source:

```
void interrupt 0 ResetFunction(void) {
  /* reset handler */
}
```

The Compiler uses the following translation from interrupt vector number to interrupt vector address:

Vector Number	Vector Address	Vector Address Size
0	0xFFFE, 0xFFFF	2
1	0xFFFC, 0xFFD	2
2	0xFFFA, 0xFFB	2
...	...	...
n	0xFFFF - (n*2)	2

Segmentation

The Linker supports the concept of segments (see Linker), i.e. 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.

Since there are two basic types of segments, code and data segments, there also are two pragmas to specify segments:

```
#pragma CODE_SEG [NEAR|FAR|SHORT] <name>
#pragma DATA_SEG [DPAGE|PPAGE|EPAGE|SHORT] <n>
```

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
```

The additional keyword `SHORT` can be used to tell the Compiler that a data segment will be allocated in the zero page (address range from 0x0000 to 0x00FF):

```
#pragma DATA_SEG SHORT <segment_name>
```

or

```
#pragma DATA_SEG SHORT DEFAULT
```

Using the zero page enables the Compiler to generate much denser code because DIRECT addressing mode can be used instead of EXTENDED.

Note: It's the programmer's responsibility to actually allocate `SHORT` segments in the zero page in the Linker parameter file. For more information, see Linker.

The *_OVERLAP* Segment (Compiler Option -Cv)

There is a special segment named `_OVERLAP` introduced by the Compiler. It only exists, if command line option –Cv is given and always must be allocated in the direct page.

If command line option –Cv is given, the Compiler is free to allocate local variables of size 2 bytes (signed or unsigned `int` or pointers) as static data in the `_OVERLAP` segment instead of on the stack. This makes accesses to these variables much faster, because they can be fetched from direct page and there are fewer instructions.

Note: For local variables that are smaller than 2 bytes this doesn't pay off, these variables can be fetched from the stack fast enough.

Of course this feature has to be used with caution: using global instead of stack memory for local variables of a certain function makes this function *non–reentrant!* Such a function must not be recursive and also should not be stopped by an interrupt unless the interrupt routine saves and restores the global memory in the `_OVERLAP` segment.

For a detailed description how data is allocated in the `_OVERLAP` segment, see

Linker.

To see the effect of using –Cv (and the _OVERLAP segment), consider the following function:

```
void any_function (void) {
  int a, b;
  char *p;
  ...
  a = b + 16;
  *p = 2;
}
```

In this case the -Cv flag makes a big difference:

```
i := a[j+2];
```

Without –Cv Switch	With –Cv Switch
`LDA b:1`	`LDHX b`
`ADD #16`	`AIX #16`
`STA a:1`	`STHX a`
`CLRA`	
`ADC b,0`	
`STA a,0`	
`LDA p:0`	`LDHX p`
`PSHA`	`LDA #2`
`LDX p:1`	`STA ,X`
`PULH`	
`LDA #2`	
`STA ,X`	

The *_DATA_ZEROPAGE* Segment

The Compiler defines another special segment (_DATA_ZEROPAGE) containing some global data space used by certain runtime support routines. As its name indicates, this segment must be allocated in direct page, too (RAM). See also Linker.

Optimizations

The Compiler applies a variety of code improving techniques commonly subsumed under the term "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 `CMP #0` or using `COMA` instead of `EORA #0xFF`.

Strength Reduction

Strength reduction is an optimization that strives to replace expensive operations by cheaper ones, where the cost factor can be either execution time or code size. Examples are the replacement of multiplications and divisions by constant powers of two with left or right shifts.

Shift Optimizations

Shifting a byte variable by a constant number of bits is analyzed intensively. The Compiler always tries to implement such shifts in the most efficient way. As an example, consider the following:

```
char a, b;
a = b << 4;
```

which results in the following code:

```
LDA b
NSA        ; Swap nibbles...
AND #0xF0  ; ...and mask!
STA a
```

Accessing Bitfields

Accesses to bitfields are inherently rather inefficient due to the mask and shift operations involved in (un)packing a bitfield. However, accesses to bitfields only 1 bit wide very often can get translated fairly well using the instructions `BSET`/`BCLR` and the bit branches `BRSET` and `BRCLR`, if the addressing mode is direct.

To enable direct addressing the bitfield must be declared in the zero page using the pragma _DATA_ZEROPAGE.

Example:

```
#pragma DATA_SEG _DATA_ZEROPAGE;
struct {
  int a:1;
} bf;

void main(void) {
  bf.a = -1;
```

```
}
```

Branch Optimizations

This optimization tries to minimize the span of branch instructions, or in other words: the Compiler will replace a relative branch by a branch with the inverted condition across an unconditional jump, if the offset of this branch is not in the range [–128 .. 127]:

```
  BRcc dest
  ...; More than 127 bytes of code
dest:
```

is changed to (provided the code indicated by "..." doesn't contain another branch to label `dest`, so that it might be possible to have the `BRcc` jump to that branch):

```
  BR~cc skip
  JMP   dest
skip:
  ...    ; More than 127 bytes of code
dest:
```

Also, branches to branches may be resolved into two branches to the same target and redundant branches (e.g. a branch to the instruction immediately following it) may be removed.

Optimization for Execution Time or Code Size

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 will generally optimize on code size; it often has to decide between a runtime routine and expanded code. In these cases, the runtime routine is chosen only if it is at least 3 bytes shorter than the expanded instruction sequence.

Register Tracing

Register tracing keeps track of what is in which register and uses the register whenever this is favorable. The Compiler does a simple global flow analysis to do effective register tracing. To show the effects of register tracing, consider the following example:

```
int a, b;
char c, d, e;
b = a;
c = e + a;
d = c + c + e;
```

The resulting code might be:

```
LDA a
STA b
LDA a:1
STA b:1
ADD e
STA c
ADD c
ADD e
STA d
```

Lazy Mode (Option -Or)

Register tracing in the Compiler has been implemented based on conservative assumptions to guarantee correct code in all cases. In most cases however, relaxing these assumptions would allow even more optimizations while still resulting in correct code. This is why there is a "lazy register optimization mode". For most programs the resulting code will be correct, only in pathological cases incorrect code will result (mainly in programs containing pointers pointing to themselves).

To switch on lazy register tracing, use the command line option “–Or”.

#pragma SAFE_OPT_ONLY

To switch off lazy register optimization for one procedure, use the pragma `SAFE_OPT_ONLY`.

To show the effects of *lazy* register tracing, consider the following example:

```
typedef struct {
  char a, b, c, d;
} *PtrType;
PtrType p;
p->a = 1;
p->b = i;
p->c = p->d;
```

The resulting code is

```
LDA   #1
LDHX  p
STA   0,X
LDA   i
LDHX  p
STA   1,X
LDHX  p
LDA   3,X
STA   2,X
```

Lazy Mode Turned **Off**

```
LDA   #1
LDHX  p
STA   0,X
LDA   i
STA   1,X
LDA   3,X
STA   2,X
```

Lazy Mode Turned **On**

With lazy mode turned on, `p` isn't reloaded every time, although basically `p` might point to itself, in which case this would be wrong because after the 3rd instruction, `p` would have changed.

Volatile Objects, Absolute Variables

The Compiler does not do register tracing on volatile global objects or absolute variables. Thus it is guaranteed that accesses to volatile global objects or absolute variables are not eliminated.

Optimization of Pointer Additions (Option -Ca)

The –Ca flag can be set, in order to optimize pointer additions. If this flag is set, the Compiler assumes, that pointer offsets (after multiplication with the object size) lie within the range

```
0 <= offset <= 0x7F,
```

e.g. for a pointer to an `int`, the Compiler assumes all non–constant offsets are in the range

```
0 <= offset <= 0x3F
```

since an `int` occupies two bytes and the offset therefore has to be multiplied by 2 before adding it to the pointer.

This has two effects. First, the addition in `A` register is faster, and second it is possible to use runtime routine `_AIX` if the pointer is in the `H:X` register.

The following is an example for the –Ca flag.

```
c =  *(p + i)
```

translates into the two different instruction sequences (assuming `p` is in the direct page and has type `char *`).

```
LDA  p:1
ADD  i:1
PSHA
LDA  p:0
ADC  i:0
PSHA
PULH
PULX
...............
LDA  ,X
STA  c
```

Without –Ca Switch

```
LDHX p
LDA  i:1
JSR  _AIX_RUNTIME
     /* fast! */




...............
LDA  ,X
STA  c
```

With –Ca Switch

The –Ca flag only has an effect on variable (i.e. non–constant) offsets. If the offset is constant, the Compiler knows its value and thus has enough information to generate the appropriate code (using the `AIX` instruction, if the constant is smaller than `0x7F`). With constant values the –Ca flag can always be used, e.g. `p - 1` is still valid with the –Ca switch active.

Since array indexing in C may involve pointer additions, the option –Ca also has an effect on array indices.

Generating Compact code With the CHC08 Compiler

Some compiler options or usage of SHORT segments can help you to generate compact code with the CHC08 compiler:

Compiler Options

Usage of following compiler options will help you to reduce the size of the code generated;

-Or: Lazy Register Optimization

When accessing pointer fields, this option will prevent the compiler to reload the address of the pointer for each access. The index register X will be loaded once and

all the fields are accessed using the addressing mode IND-X.

-Cni: Non integral promotion on integer

When this option is specified, the ANSI C integral promotion does not apply to character comparison or arithmetic. This can same a big amount of code.

-Oc: Common subexpression Elimination

SHORT Segments

Variables allocated on the direct page (between 0 and 0xFF) can be accessed using the direct addressing mode. You can tell the compiler that some variable are allocated on the direct page, if you define them in a SHORT segment.

Example:

```
#pragma DATA_SEG SHORT 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 segment, the external declaration for these variables must also specify that they are allocated in a SHORT segment. The External definition of the variable defined above will look like:

```
#pragma DATA_SEG SHORT myShortSegment
extern unsigned int myVar1, myVar2;
#pragma DATA_SEG DEFAULT
extern unsigned int myvar3, myVar4.
```

Then the segment must be placed on the direct page in the PRM file.

Example:

```
LINK test.abs
NAMES test.o start08.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;
    _DATA_ZEROPAGE, myShortSegment INTO  Z_RAM;
```

```
END
STACKSIZE 0x60
VECTOR ADDRESS 0xFFFE _Startup /* set reset vector on _Startup */
```

Be careful: The linker is case sensitive. The segment name must be identical in the C and PRM file.

Defining IO Registers

HC08 IO Registers are usually base at address 0. In order to tell the compiler it should use direct addressing mode to access the IO registers, these registers can be defined in a SHORT section based at the specified address.

In the C source file, the IO register are defined 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 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 start08.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;
    _DATA_ZEROPAGE                 INTO  Z_RAM;
    SCIRegs                        INTO  SCI_RG;
END
STACKSIZE 0x60
VECTOR ADDRESS 0xFFFE _Startup /* set reset vector on _Startup */
```

Be careful: The linker is case sensitive. The segment name must be identical in the C and PRM file.

High Level Inline Assembler for Motorola HC08

The HLI (High Level Inline) Assembler provides a comfortable means to make full use of the properties of the target processor right within a C program. There's no need to write a separate assembly file, assemble it and later bind it with the rest of the application, which is written in ANSI-C/C++ with the inline assembler, the Compiler does all that work! For further information, please refer to the HC08 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:

```
"asm" <Assembly Instruction> ";" ["/*" Comment "*/"]
"asm" <Assembly Instruction> ";" ["//" Comment]
```

or

```
"asm" "{"
 { <Assembly Instruction> [";" Comment] "\n" }
"}"
```

If you use the first form, multiple `asm` statements can be on one line and comments are delimited like regular C or C++ comments. If you use the second form, one to several assembly instructions can be within the `asm` block, but only one assembly instruction per line is possible and the semicolon starts an assembly comment.

Inline Assembly Language

First, we give the EBNF Syntax of an inline assembly instruction. A short discussion on semantics and special constructs follows.

```
AsmLine       = [Label](Code | Directive).
Label         = ident ":".
Code          = mnemonic [ArgList][OptParams].
Directive     = Code.
ArgList       = Argument {"," Argument}.
```

```
Argument      = Imm | Expression.
Imm           = "#" Expression.
OptParams     = "!" RegList "," RegList.
RegList       = "{" [Reg] {"," Reg} "}".
Reg           = "A"|"HX"|"H"|"X"|"SR".
Variable      = Identifier.
Expression    = [Term {("+"|"-") Term}] ["," ("X"|"HX"|"SP"|"FP")].
Term          = Factor {("*"|"/") Factor}.
Factor        = "(" Expression ")"
                | "-" Factor
                | "@" Factor
                | "*" Factor
                | Factor {":" Factor|"MSB")}
                | Number
                | TypeQualifier
                | Variable {"." Field}
                | Procedure
                | Label
TypeQualifier = Type "." Field {"." Field}.
Procedure     = Identifier.
Label         = Identifier.
Type          = Identifier.
Field         = Identifier.
```

To resolve ambiguities of the grammar, the longest possible match is taken to reduce to Factor (shift is always preferred). The pseudo offset `MSB` designates the most significant byte of an effective address; it is only allowed for memory operands. The `TypeQualifier` production above evaluates to the offset of the given field. `Type` must be the name of a `struct` type, and all field names given must exist.

Examples

A short and simple example shall illustrate the use of the HLI-Assembler.

Assuming that `str` points to a character array, we can write a simple function in assembly language to determine the length of a string:

```
int strlen (char *str)
  /*** 'str' is passed on stack. Returns length of string 'str'.
       This procedure assumes len(str) is smaller than 256! */
{
  asm {
    LDHX str       ; load pointer
    CLRA           ; init counter
    BRA test       ; go to test
  loop:
    AIX  #1        ; increment pointer
```

```
      INCA           ; increment counter
    test:
      TST 0,X        ; not end of string?
      BNE loop       ; next char
      CLRX           ; return value in X:A(see later)
    };
    /* C statements could follow here */
  }
```

Note: If pragma NO_ENTRY is not set, the Compiler takes care of entry and exit code, the programmer doesn't have to worry about setting up a stack frame him-/herself.

Special Features

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 (`LDA, STX, ...`)
- all register names (`A, H, X, HX, SR, SP`)
- the identifier `MSB`

For these reserved words, the inline assembler is *not* case sensitive, i.e. `LDA` is the same as `lda` or even `LdA`. For all other identifiers (labels, variable names and so on) the inline assembler is case sensitive.

Pseudo-Opcodes

The inline assembler provides some pseudo opcodes to put constant bytes into the instruction stream. These are:

```
DCB 1      ; Byte constant 1
DCB 0      ; Byte constant 0
DCW 12     ; Word constant 12
DCL 20, 23 ; Longword constants
DCF 1.85   ; IEEE 32bit float
DCD 2.0    ; double IEEE64
```

Accessing Variables

The inline assembler allows accessing local and global variables declared in C –

simply use their name in the instruction.

Address Notation

In addition to the address syntax described in the CPU08 manual, one may access objects using a C–like syntax and symbolic variable names. In this case all addresses are assumed to be of type `char *`.

```
@var       ==> address of var
*var       ==> object pointed to by var
var:offs   ==> *((@var) + offs)
obj.field  ==> *((@obj) + offset_of(field))
type.field ==> offset_of(field)
```

H:X Instructions

`LDHX` and `STHX` are allowed with any reasonable argument, and are not restricted to DIRECT page and IMMEDIATE arguments any more. They represent code patterns which are automatically translated into an appropriate code sequence. Consider the following example (assume `local` is a variable on the stack):

Inline Assembly	Expansion

`LDHX  local`	`LDA  local:0`
	`LDX  local:1`
	`PSHA`
	`PULH`

`LDHX  @local`	`TSX` `AIX  #<local-offset>`

Constant Expressions

Constant expressions may be used anywhere an IMMEDIATE value is expected. They may contain the binary operators for addition ("+"), subtraction ("-"), multiplication ("*"), and division ("/"). Also, the unary operator "–" is allowed. Round brackets may be used to force an evaluation order other than the normal one. The syntax of numbers is the same as in ANSI-C.

A constant expression may also be the address of a global variable or the offset of a local variable.

Optimizing Inline Assembly

The Compiler is free to modify the instruction stream as long as its semantics remains the same. This is especially true for stack pointer indirect accesses (`ind`, `SP`, e.g. accesses to local variables), which may be replaced by `TSX` and a `ind`, `H:X` access.

Assertions

The pseudo instruction

```
_ASSERT  <reg>, addr
```

declares that the register <`reg`> (one of `H`, `X`, `HX`, `A`) contains `addr` at this instruction. It may be used by the Compiler to perform optimizations, but code will still be correct if this information is ignored. A typical example might be the following:

```
    LDHX @a:4
    MOV  #4, tmp
loop:
    AIX #-1
    LDA 0,X
    STA 4,X
    DBNZ tmp, loop
    _ASSERT HX, @a
```

Stack Adjust

The pseudo instruction

```
_STACK  label:offs
```

declares that the stack pointer's value at this instruction is (stack pointer's value at `label`) + `offs`. It is never ignored by the Compiler, since the stack pointer may be used to access local variables.

The pseudo instruction

```
_ADJ  label:offs
```

adjusts the `SP` register with appropriate `AIS` instructions, so that `_STACK` `label:offs` holds afterwards. The `_STACK` instruction is useful for situations like the following:

```
    base CLRA
    MOV  #8, tmp
loop:
    PSHA
```

```
DBNZ  tmp, loop
_STACK base:-8
```

In & Gen Sets

The pseudo instruction

```
_INGEN  <in-set>, <gen-set>
```

announces to the Compiler that registers in <in-set> are used further, and those in <gen-set> are not. The notation

```
<instr> <args> ! <in-set>, <gen-set>
```

allows to declare which registers are used, and which ones are modified by an instruction. This is very sensible for JSR, JMP, RTS, .. instructions. The default is always on the conservative side. Consider the following example:

```
LDA a:1
ADD b:1
STA b:1
LDA a:0
ADC b:0
STA b:0
RTS ! {},  {}
```

This allows the Compiler to destroy the `H:X` register for accesses to local variables. With the simple `RTS`, the Compiler does not know if the user expects the `H:X` register to be preserved or not.

Multiple registers must be separated by a '+'. For example :

```
RTS ! {A+X},  {}
```

The following examples illustrate the usage with the original code and the generated one:

```
char f(void);

char asm1(int a) {
  int b;
  asm {
     LDA  a:1
     STA  b:1
     JSR  f ! {}, {}
     STA  a:0
     LDA  b:1
     RTS  ! {}, {A}
  }
}
```

```
char f(void);

char asm1(int a) {
  int b;
  asm {
     TSX
     LDA 5,X  ; H:X can be used
     STA 1,X  ; to access all locals
     JSR f
     STA 4,X
     LDA 1,X
     RTS
  }
```

```
                                }

char asm2(int a) {              char asm2(int a) {
  int b;                          int b;
  asm {                           asm {
     LDA  a:1                        LDA 6,SP ; X is input register
     STA  b:1                        STA 2,SP ;  for f. SP must be used
     JSR  f ! {X}, {}                JSR f
     STA  a:0                        TSX      ; f does not set X.
     LDA  b:1                        STA 4,X  ; now X is free
     RTS  ! {}, {A}                  LDA 1,X
  }                                  RTS
}                                 }
                                }

char asm3(int a) {              char asm3(int a) {
  int b;                          int b;
  asm {                           asm {
     LDA  a:1                        TSX
     STA  b:1                        LDA 5,X   ; H:X can be used
     JSR  f ! {}, {X}                STA 1,X   ; to access locals
     STA  a:0                        JSR f     ; f destroyes X
     LDA  b:1                        TSX       ; X is restored
     RTS  ! {}, {A}                  STA 4,X   ; to access locals
  }                                  LDA 1,X
}                                    RTS
                                  }
                                }

char asm4(int a) {              char asm4(int a) {
  int b;                          int b;
  asm {                             asm {
     LDA a:1                          LDA 6,SP
     STA b:1                          STA 2,SP
     JSR f ! {}, {}                   JSR f
     STA a:0                          STA 5,SP
     LDA b:1                          LDA 2,SP
     RTS ! {X}, {A}                   RTS ; X must not be used
                                          ; in this function!
  }                                 }
}                               }
```

Entry, Exit

The pseudo instruction

```
_ENTRY
```

announces to the Compiler that this instruction may be jumped to externally (or by

some mechanism other than a branch to label).

The pseudo instruction

```
_EXIT
```

announces to the Compiler that the instruction sequence is left here. Application of this feature is rare, but the example below shows a possible usage.

```
            ; implicit parameter passing at different entry points
            ;
in_0:       ; start of procedure is default entry
            ;
    LDA #1
    BRA begin
in_1:       ;
    _ENTRY
    LDA #2
            ;
begin:      ; procedure begins with argument in A
    ...
```

When using _ENTRY, do not forget to adjust the stack if necessary. The following shows the application of _EXIT:

```
    ; fast return mechanism
    ; 'return_float' gets constant from code
    ; and returns directly
    ...
ret_0: JSR return_float
    DCF 1.0
   _EXIT
ret_1: JSR return_float
   DCF 2.0
  _EXIT
```

Optimization Info

Pseudo instruction

```
_OPT_RA <effective address>
```

tells the Compiler where the return address of the current procedure can be found, and is used for special run-time routines, which move the return address around. The default address is `0, FP`; other typical addresses are `off, SP`, some global variable or the `H:X` register contents. Some examples:

```
    ; assume return address is TOS
    ;
PULH
```

```
PULX
_OPT_RA HX  ; HX contains the return address
STHX tmp
_OPT_RA tmp ; tmp now containsthe return address
```

Getting the High Address Part in HLI

It is possible to get the high address part of an object using the MSB syntax. The following short example illustrates this:

```
int i, *p_i;

void main(void) {
  /* p_i = &i */
  asm {
          LDA @i
          STA p_i:1
          LDA @i:MSB
          STA p_i
  }
}
```

Note: R1 is designated as a scratch register upon entry to a subroutine to calculate stack frame adjustments for subroutines with large stack frames. It is also useful for saving and restoring the non-volatile registers at function entry and exit.

Messages

This chapter describes the messages produced by the Compiler. Because of the huge amount of different messages produced by the Compiler, not all of them may be in this document yet.

Message Kinds

There are five kinds of messages generated by the Compiler:

INFORMATION

A message will be printed and the compilation will continue. Information messages are used to indicate actions taken by the Compiler to the user.

WARNING

A message will be printed and the compilation will continue. Warning messages are used indicate possible programming errors to the user.

ERROR

A message will be printed and compilation will be stopped. Error messages are used to indicate illegal usage of the language.

FATAL

A message will be printed and compilation will be aborted. A fatal message indicates a severe error which anyway will stop the compilation.

DISABLE

The message has been disabled. No message will be issued and compilation will continue as if there was no such message, the Compiler just ignores this kind of message.

Message Details

If the Compiler prints out a message, the message contains a message code (‘C’ for

Compiler) and a four to five digit number. This number may be used to search very fast for the indicated message.

All messages generated by the Compiler are documented in increasing number 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.

Some versions of the Compiler are also capable to change the type of a message, e.g.

[DISABLE, INFORMATION, WARNING, ERROR]

indicates that the message is a warning message by default, but the user might change the message either to DISABLE, INFORMATION or ERROR.

Message List

The following pages describe all Messages.

All message number below 10000 are common to all HICROSS compilers. Therefore not every compiler does emit all of them. For example many compiler do support any type of struct return. Such compilers will never emit the message “C2500: Expected: No support of class/struct return type”.

Also many messages are C++ specific. These messages will only be emitted from C++ compilers.

C1: Unknown message occurred

[FATAL]

Description:

The compiler 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 compiler 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 messag-

es.

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 handles 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:\hicross\lib\
OBJPATH=c:\hicross\work
...
```

Is taken by the compiler identical as

```
...
LIBPATH=c:\hicross\libOBJPATH=c:\hicross\work
...
```

Tips:

To fix it, append a '.' behind the '\'

```
...
LIBPATH=c:\hicross\lib\.
OBJPATH=c:\hicross\work
...
```

Note:

Because this information occurs during the initialization phase of the compiler, the 'C' might not occur in the error message. So it might occur as "64: Line Continuation occurred in <FileName>".

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 globals
```

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 globals
```

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! (There should be no error in later releases!).

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
}
```

C1014: Integral type expected or enum value out of range

[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.

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.

Example:

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.

Example:

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.

Example:

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.

Example:

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).

Example:

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, Assign-

ment operator) was created by the compiler.

Example:

Tips:
If you do not want the compiler to generate the special member functions, then enable the option -Cn=Ctr.

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.

Example:

Tips:

Depending on FunctionKind:
- operator -> 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.

Example:

Tips:

Depending on FunctionKind:
- operator -> 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 FunctoinKind has a parameter of a wrong type.

Example:

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

Example:

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.

Example:

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).

Example:

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).

Example:

Tips:
Do not use the storage classes ”static” and ”extern” for destructors.

C1069: Wrong use of far/near/rom/uni/paged in local scope

[ERROR]

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.

Example:

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 ().

Example:

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.

Example:

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.

Example:

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.

Example:

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.

Example:

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.

Example:

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.

Example:

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.

Example:

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.

Example:

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.

Example:

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.

Example:

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.

Example:

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;
```

Tips:
The index value must be greater than 0.

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 parameters.

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 backend do support non int bitfields, other do not. See the chapter backend for details.

Example:

```
struct A {
  char i:2;
}
```

When the actual backend does support 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 Constructor
}
```

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 hill-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:
1.) Use a pointer/reference to the object:

```
void main(void){
  A *pa;
}
```

2.) 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:

1.) Use a pointer/reference to the object:

```
void main(void){
  void fct(A *);
}
```

2.) 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:

1.) Use a pointer/reference to the object

```
void main(void){
  A *fct(void);
}
```

2.) 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 a 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.

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 -Qvtprom 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 -Qvtprom.

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 setup 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 an other
```

```
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;
}
```

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 is not yet implemented

[ERROR], C++

Description:

The C++ compiler does not yet support all C++ features. See also C++ section about not-yet implemented features.

Tips:

Try to find a workaround of the problem or ask about the latest version of the compiler.

C1402: This C++ feature is not yet implemented

[ERROR], C++

Description:

The C++ compiler does not yet support all C++ features.
Here is a list of not yet supported C++ features causing an 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 the not-yet implemented 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.

Example:

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

[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.

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.

Example:

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 nonvoid parameter
    A();      // default constructor
};
```

Tips:
If you provide a constructor that takes a nonvoid 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.

Example:

Tips:
Initialize the member inside the function body of the constructor.

C1425: No Destructor available to call

[ERROR], C++

Description:
No destructor is available, but one must be called.

Example:

Tips:

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.

Example:

Tips:
Do not use ”this” in global functions.

C1428: No wide characters supported

[DISABLE, INFORMATION, WARNING, ERROR]

Description:
The Compilers 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.

Example:

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 provide 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)();
}
```

C1800: Implicit parameter-declaration (missing prototype)

[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();
void g(void);
```

In C the declaration of 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

[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.

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.

Example:

Tips:

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.

Example:

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:

Tips:

Use ”if-else if” statements instead.

C1809: Integer value for switch-expression expected

[ERROR]

Description:

The specified switch expression evaluated to an illegal type.

Example:

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.

Example:

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.

Example:

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!.

Example:

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:

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:

Tips:
On the right side of the "->" or "." operator, there must be a member of the structure/union specified on the left side.

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:

Tips:
Don’t 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.

Example:

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:

Tips:

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 nonpointer value.

Example:

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:

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:

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.

Example:

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:

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.

Example:

Tips:

C1840: Pointer to struct or union expected

[ERROR]

Description:

A pointer to a structure or union was expected.

Example:

Tips:

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..

Example:

Tips:

Use an integer type in the switch expression..

C1844: Call-operator applied to non-function

[ERROR], C++

Description:

A call was made to a function through an expression that did not evaluate to a function pointer.

Example:

Tips:

The error is probably caused by attempting to call a nonfunction.

C1845: Constant integer-value expected

[ERROR]

Description:

The case expression was not an integral constant.

Example:

Tips:
Case expressions must be integral constants.

C1846: Continue outside of iteration-statement

[ERROR]

Description:
A “continue” was made outside of an iteration-statement.

Example:

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:

Tips:
The “break” must be done inside an iteration-statement..

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.

Example:

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:

Tips:
Don’t 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.

Example:

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 formal parameters).

Example:

Tips:

Both operands of the binary operator must have compatible types.

C1852: Illegal sizeof operand

[ERROR]

Description:

The sizeof operand was a bitfield.

Example:

Tips:

Don't 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 a unsigned int, if int is 32 bits large or a 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.
So, the value -2147483648 is the negation of 2147483648 and therefore also of a unsigned type, even if the signed representation does contain 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.

C2000: No constructor available

[ERROR],C++

Description:

A constructor must be called, but none is available.

Example:

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.

Example:

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.

Example:

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 template parameter didn't occur in the parameter list of the template func-

tion.

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:
Don't 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.

Example:

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
```

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.
Or compile your code with C++, so this error won’t occur anymore!

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.

Example:

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

C2212: Global variable initialization is not supported yet

[ERROR], C++

Description:

Initialization of global pointer to member is not supported yet by our compiler.

Example:

```
#pragma A
#pragma B
#pragma C
...
```

Tips:

Remove all pragmas not known to the compiler.
The compiler can handle much more pragmas that he knows.
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 HICROSS 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.
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

[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 do 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.

Example:
```
signed char ch = 128; /* char holds only -128..127 */
```

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.

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 do 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.

Example:
```
int $j;
```

Tips:

Usually this is a programming error. Replace the illegal character with a legal one.

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 will be 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 will be removed by the compiler and only the code for the initialization 'i=10' will be 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:

L0oop Unrolling

C3000: File-stack-overflow (recursive include?)

[FATAL]

Description:

There are more than 32 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 32 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 does occur for Modula 2 versions of the compiler only. They do not occur 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 then 400'000 lexical tokens.

See also:
Limitations

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 then 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 2048 characters.

Example:
Implicit string concatenation of two strings with each more than 1024 characters:

```
char *str = "MoreThan1024...." "OtherWithMoreThan1024...."
```

Tips:
Do not use implicit string concatenation, write the string in one piece:

```
char *str = "MoreThan1024....OtherWithMoreThan1024...."
```

See also:
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 will be 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 the programmer does not like 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.

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)

[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;
```

See also:

Limitations

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.

Tips:

Report this error to your distributor.

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 do have at least a return instruction, this error can only occur with the pragma NO_EXIT. Remark that not all targets do 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 pragma TEST_CODE is for internal use only. It is used to compare the code produced by the current compiler with the code of a previous compiler.

Example:

```
#pragma TEST_CODE == 1
void main(void) {}
```

Tips:

Do not use this pragma. It is for internal use only.

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 the 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 guarantee 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 guarantee 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

[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.

C3803: Illegal Segment Attribute

[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.

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 will implicit convert a shift count larger than 8bit (char) to a unsigned char, avoiding to load a shift count too large for shifting which anyway does not affect the result.
This guarantees 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 compile 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

[DISABLE, INFORMATION, WARNING, ERROR]

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->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 a unsigned bitfield entry or compare the value with -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 do never issue this warning, even if they are not used.

Example:

```
#pragma TRAP_PROG
  /* tipping 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

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.

C4400: Comment not closed

[FATAL]

Description:
The preprocessor has found a comment which 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 '>'.

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

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.

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
#if UNDEFINED_MACRO /* evaluates to 0, giving this warning */
#endif
#if DEFINED_MACRO   // error FATAL: C4435: Illegal expression in
                    // conditional expression */
#endif
```

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: Error <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.

Example:

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.

Example:

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 compiler internal error happened!

Tips:
Please contact your distributor.

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 HI-CROSS Object File Format

[ERROR]

Description:
For HICROSS 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 ever if the user only need multibyte characters.
For the strict HICROSS 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 HI-CROSS 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 HI-CROSS 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:

Do use -Fh HICROSS object file format to change the enum type. To change also the char type, only 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

C5300: Limitation: code size > 'Size' bytes

[ERROR]

Description:
You have a limited version of the compiler (demo version).

Tips:
Get a license for a full version of the compiler!

C5301: Limitation: too many procedures (>'Number')

[ERROR]

Description:
You have a limited version of the compiler (demo version).

Tips:
Get a license for a full version of the compiler!

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 do also 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 do also report errors for which you do have a work around.

See also
Chapter Bug Report

C5702: <Variable>: local 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.
This message is only used for HICROSS + compilers.

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 will be 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 will be 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 will be 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 treats this as an error.
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 will be 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 will be 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 will be 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 will be 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) {  // ... and 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) {// ... and 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(!ondition) { // ... and 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) { // ... and 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.

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 of 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.
E.g.
```
#pragma CREATE_ASM_LISTING ON
```
or
```
#pragma CREATE_ASM_LISTING OFF
```

See also :
Option -La
Create Assembler Include Files
pragma CREATE_ASM_LISTING

ANSI Library Reference

Library Files

Directory Structure

The library files are delivered in a new structure:

```
<install>\lib\<target>c\        /* readme files, make files */
<install>\lib\<target>c\src     /* library source files (C and C++)*/
<install>\lib\<target>c\include /* library include files */
<install>\lib\<tartet>c\lib     /* default library files */
<install>\lib\<target>c\lib.hix /* HI-CROSS library files */
<install>\lib\<target>c\lib.e11 /* ELF/DWARF 1.1 library files */
<install>\lib\<target>c\lib.e20 /* ELF/DWARF 2.0 library files */
```

Previously everything was installed in '<installation>\lib\hc12c\.' The new scheme allows a more structured library usage than before.

The installation script adds '<install>\lib\<target>c\', '<install>\lib\<target>c\src' and '<install>\lib\<target>c\include' to the GENPATH environment variable in the default.env file. It adds only '<install>\lib\<target>c\' and '<install>\lib\<target>c\include' to the LIBPATH environment variable in the default.env file. This allows compatibility with older installations.

Not all object file formats (HI-CROSS, ELF/DWARF 1.1, ELF/DWARF 2.0) my be delivered with the installation and also the default format may differ. To change the default object file format just two directories have to be renamed. E.g. to change from a ELF/Dwarf default format to the HI-CROSS one, you just have to rename two directories:

rename <installation>\lib\hc12c\lib to <installation>\lib\hc12c\lib.e20

rename <installation>\lib\hc12c\lib.hix to <installation>\lib\hc12c\lib

To use the new structure in your own default.env files, just add the additional paths to your GENPATH and LIBPATH environment variables. Do not forget to change the object file format option (-Fh, -F1, -F2) to the correct one and also to use the right linker (linker.exe (HI-CROSS) and elflinker.exe (ELF/Dwarf)) and the right decoder (decoder.exe (HI-CROSS) and elfdecod.exe (ELF/Dwarf)).

Common Source Files

The source and header files of the Standard ANSI Library which are not target dependent are in the following table.

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.

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.

Every function in the library can be overwritten by simply including the redefinition ahead of the library in the linker parameter file (.prm)

The following libraries will be on your distribution disk.

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 reentrant, 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.

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.

1. A buffer is needed which alone may use a large amount of RAM.

2. 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 `vsscanf`.

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.

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 * strpbrk (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 microcontroller 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.

`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`.

`DIG`	Number of significant digits.
`EPSILON`	Smallest positive `x` for which `1.0 + x != x`.
`MANT_DIG`	Number of binary mantissa digits.
`MAX`	Largest normalized finite value.
`MAX_EXP`	Maximum exponent such that $\texttt{FLT_RADIX}^{\texttt{MAX_EXP}}$ is a finite normalized value.
`MAX_10_EXP`	Maximum exponent such that $\texttt{10}^{\texttt{MAX_10_EXP}}$ is a finite normalized value.
`MIN`	Smallest positive normalized value.
`MIN_EXP`	Smallest negative exponent such that $\texttt{FLT_RADIX}^{\texttt{MIN_EXP}}$ is a normalized value.
`MIN_10_EXP`	Smallest negative exponent such that $\texttt{10}^{\texttt{MIN_10_EXP}}$ is a normalized value.

limits.h

Defines a couple of constants for the maximum and minimum values that are allowed for certain types.

`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

This header file defines a struct containing all the locale 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:

`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 only one constant:

`HUGE_VAL` Large value that is returned if overflow occurs.

setjmp.h

Contains one type definition:

`typedef jmp_buf;` A buffer for `setjmp` to store the current program state.

signal.h

Defines signal handling constants and types.

`typedef sig_atomic_t;`

`SIG_DFL` If passed as 2nd 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.

The following signals exist:

`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.

`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.

`#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:

`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`.

Also, there are a couple of constants:

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 following types:

typedef div_t;	A struct for the return value of div.
typedef ldiv_t;	A struct for the return value of ldiv.

Additionally, it contains these constants:

EXIT_FAILURE	Exit code for unsuccessful termination.
EXIT_SUCCESS	Exit code for successful termination.
RAND_MAX	Maximum return value of rand.
MB_LEN_MAX	Maximum number of bytes in a multi byte character.

time.h

This header files defines types and constants for time management.

```
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 do 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 file ctype.h 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 do 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 do additionally save memory by the often shorter call to the function compared with the expanded macro.

The functions are explained below together with all other ANSI functions.

```
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 */

#ifdef __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 following subsections:

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 `errno` 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 that 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 <= x <= 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, mktime, 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 <= x <= 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 `Pi` 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

Parameter Name	Meaning
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()`.

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 `mktime`, `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: Don't 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 *not* assume the value in `*pos` 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 `*pos` 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, cnst 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 of 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.

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 allocates by `calloc`, `malloc` or `realloc`. If `ptr` is `NULL`, nothing happens.

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 * 2^exp`. m always is normalized to the range `0.5 < m <= 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 `ftell`.

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 `fgetpos`, `fsetpos`, `ftell`

fsetpos

File I/O

Syntax

```
#include <stdio.h>
int fsetpos (FILE *f, const fpos_t *pos);
```

Description `fsetpos()` set the file position to `pos`, which must be a value returned by a previous call to `fgetpos` 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 `fgetpos`, `fseek`, `ftell`

ftell

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 `ch` is in the range...
isalnum	alphanumeric character, i.e. `'A'–'Z'`, `'a'–'z'` or `'0'–'9'`.
isalpha	an alphabetic character, i.e. `'A'–'Z'` or `'a'–'z'`.
iscntrl	a control character, i.e. `'\000'–'\037'` or `'\177'` (`DEL`).
isdigit	a decimal digit, i.e. `'0'–'9'`.
isgraph	a printable character except space (`'!'–'~'`).
islower	a lower case letter, i.e. `'a'–'z'`.
isprint	a printable character (`' '–'~'`).
ispunct	a punctuation character, i.e. `'!'–'/'`, `':'–'@'`, `'['–'`'` and `'{'–'~'`.

Function	Tests whether ch is in the range...
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`^{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 *localetime (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);
```

Decsription `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()`.

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 mblen (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` don't 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 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`, `ldexpf`, `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.

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
array	A pointer to the beginning (i.e. the first element) of the array you want to sort
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.

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`.

Return `ralloc` 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 `strcoll` and `strxfrm`.
`LC_MONETARY`	for monetary formatting.
`LC_NUMERIC`	for numeric formatting.
`LC_TIME`	for function `strftime`.
`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 will be buffered

Mode	Buffering
_IOFBF	Fully buffered
_IOLBF	Line buffered
_IONBF	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
SIG_IGN	The signal is ignored.
SIG_DFL	The default response (`HALT`).
a function	The function is called with `sig` as parameter.

The signal handling function is defined as

```
typedef void (*_sig_func) (int sig);
```

Signal can be raised using function `raise`. Just 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 will be 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.

Conversion	Precision
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 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 intergral 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.

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`

srand

Syntax

```
#include <stdlib.h>
void srand (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 `char *`. The string read is *not* zero–terminated.
d	A decimal number (syntax below) is read an 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.

Conversion	Description
n	The argument must be a pointer to an `int`. `sscanf` 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 `short int` or a `long int`.
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 `sprintf` prints it. The parameter must be a `void **`.
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 char *.
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.
%m	Month, as a decimal number from 0 to 12.
%M	Minutes

Format	Replaced with
%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: `mktime`, `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`

strncmp

Syntax
```
#include <string.h>
char *strncmp (char *p, const char *q, size_t n);
```

Description `strncmp()` 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.

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 HI-CROSS+ 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 HI-CROSS+ 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:

random.cpp

tempbuf.cpp

streamb.h, streamb.cpp

iostream.h, iostream.cpp

cstring.h, cstring.cpp,

new.h, new.cpp

stderr.cpp

cmathi.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, stl.h, algo.h, deque.h, function.h, iterator.h, list.h, map.h, memory.h, queue.h, set.h, stack.h, vector.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 stdlog of the ANSI-C library)

These objects offer an easy way to perform input and output in a C++ program.

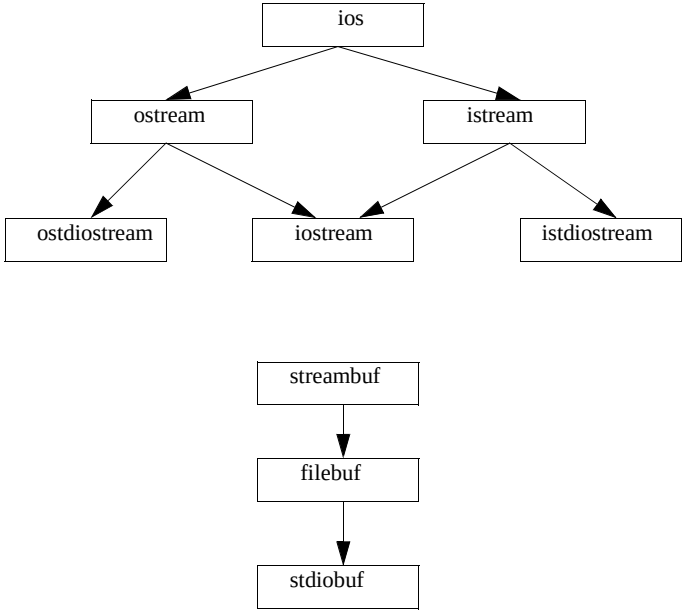
Headers

iostream.h

streamb.h

stdiostr.h

Classes



Example

```
#include “iostream.h”

void main() {
  cout << “Hello ” << “World” << endl;  // prints out “Hello World\n”
}
```

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 occured !
  }
}
```

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:

```
#include <bitset.h>
void main() {
  BitSet bs, bs2;
  bs.set(4);
  bs2.set(3);
  bs=bs|bs2;
  if(bs.test(1)!=0) {
    /* Error occured */
  }
  if(bs.test(2)!=0) {
    /* Error occured */
  }
  if(bs.test(3)==0) {
    /* Error occured */
  }
  if(bs.test(4)==0) {
    /* Error occured */
  }
  if(bs.test(5)!=0) {
    /* Error occured */
  }
}
```

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;
}
```

STL (Standard Template Library)

Description:

The Standard Template Library is a C++ library of template-based container classes, iterators, and common algorithms, such as searching and sorting.

Headers:

stl.h

algo.h

deque.h

function.h

iterator.h

list.h

map.h

memory.h

queue.h

set.h

stack.h

vector.h

Appendix

This appendix describes some FAQ's, hints how to port your application from a different tool vendor or the syntax of EBNF.

Migration Hints

This chapter describes the differences between this Compiler and the Compilers of other vendors.

This chapter provides some information about porting sources and how to adapt them.

Allocation of Bit Fields

Allocation of bitfields is very Compiler dependent: some allocate the bits first from right (LSByte) to left (MSByte), some other Compiler just the other way. Also alignment and byte/word crossing of bitfields is not implemented consistently. Check the different allocation strategies and check if there is an option to change the allocation strategy in the Compiler or use the Compiler defines to hold sources portable:

__BITFIELD_LSBIT_FIRST__

__BITFIELD_MSBIT_FIRST__

__BITFIELD_LSBYTE_FIRST__

__BITFIELD_MSBYTE_FIRST__

Type Sizes, Sign of Character

Check carefully 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 -T 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.

@tiny 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 HIWARE compiler does not need the @tiny qualifier directly, the compiler is smart enough to take the right addressing mode depending on the address:

```
volatile char REG_PTB @0x01; // compiler uses the correct
                                addressing mode
```

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 */
}
```

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.

Frequently Asked Questions (FAQ's), Trouble Shooting

In this section we'll give you some hints how to solve some of the most commonly encountered problems when working with the tools. We also tell you how to contact our Technical Support if all else fails.

Installation

Installing is such an easy task that there should be no problems at all. If you do have problems all the same, check the following points:

Do you have Windows 3.0 or higher installed?

Did you enter the correct user ID and license number?

Did you log in as super–user?

Is the tape rewound?

Did you specify the correct directories?

Do you have enough space on the disk for the installation? (Also check your quota!)

If all of these points are OK, check the distribution disk. If it can't be read, call your local distributor and ask for a new distribution disk to be sent to you. It can sometimes happen that disks become unreadable due to exposure to magnetic fields during transport.

Running Programs

If you have any problems (e.g. crashes) with running programs, first check for virus infection. If you have made sure that your computer is "clean" using an anti–virus program and the problem still remains, check the following points:

Did you run the latest version of the program?

Is your environment set up correctly?

Did you try to start the program twice?

Is the executable file corrupted? (In case of doubt, re–install the software.)

If you cannot solve the problem and it is reproducible, send a bug report with a precise description of the problem to our Technical Support Department.

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 EDOUT 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 Input Files of this document.

The Compiler Seems to Generate Incorrect Code

First make sure the code is really wrong. Sometimes the operator precedence rules of ANSI–C do not quite give the results one would expect, so apparently faulty code might 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 really 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.

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

Things you’d like to see improved in a future major release, that would be handy, but you can live (and work!) without.

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:

HIWARE REPORT FORM

(Fill this form and send it to HIWARE using:

EMail: support@hiware.hicom.ch

Fax : ++(41) 61 690 75 01

)

CUSTOMER INFORMATION

Customer Name :

Company :

Customer Number:

Phone Number:

Fax Number:

Email Address:

PROCUCT INFORMATION

Product (HI-CROSS+, HI-WAVE, Smile Line, ...):

Host Computer (PC, Sun, ...):

OS/Window Manager (Win 3.11, Win32s, WinNT, Win95, ...):

Target Processor:

Language (C, C++, ...):

TOOL INFORMATION

Tool (Compiler, Linker, ...):

Version Nr (Vx.x.xx):

Options Used:

For HI-WAVE 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 using electronic mail. 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@hiware.hicom.ch

To get information about newest updates and product enhancements, the world wide web page can be visited:

http://www.hiware.com

Sending a fax describing your problem is possible under the following number:

+41 61 690 7501

To reach technical support by normal mail, use the address below:

HIWARE AG
Technical Support
Riehenring 175
4058 Basel (Switzerland)

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. Therefore a short introduction to EBNF is given here.

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. An EBNF grammar is a set of rules called *productions* of the form:

```
LeftHandSide = RightHandSide.
```

The left hand side is a so-called non-terminal symbol, the right hand side describes how it is composed.

EBNF consists of the following symbols:

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 EBNF itself, the second one is a terminal symbol (it is quoted) and therefore may appear in the language described.

Production End

A production always is terminated by a period.

EBNF-Syntax

We can now give the definition of EBNF in EBNF itself:

```
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, we use the following notational conventions:

The counting repetition: Anything enclosed by "{" and "}" and followed by a $^{\text{super-scripted}}$ expression *x* must appear exactly *x* times. *x* may also be a non-terminal. In the following example, exactly four stars are allowed:

```
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:

```
Struct = RefNo FilePos[4].
```

In some examples, we enclose text 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 floating number suffixes are ‘f’ and ‘F’ for float and ‘l’ or ‘L’ for 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

Note that the suffixes are not case sensitive, e.g. ‘ul’, ‘Ul’, ‘uL’ and ‘UL’ all denotes 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
`== !=`	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

*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*", Sec-

ond 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;
```

Here the associativity rules cause first to evaluate c+=1, then b to be assigned 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

Index

Symbols

. 239
. . . . . 239, 269
#define 141, 239
#elif 239
#else 239
#endif 239
#error 239
#if 239
#ifdef 239
#ifndef 239
#include 138, 239
#line 239
#pragma 239
CODE_SECTION 259
CODE_SEG 211, 259, 354
CONST_SECTION 103, 259
CONST_SEG 214, 259
DATA_SECTION 259
DATA_SEG 218, 259
FAR 354
INLINE 155, 221
INTO_ROM 103, 222
LOOP_UNROLL 122, 223
MESSAGE 224
NEAR 354
NO_ENTRY 225, 367
NO_EXIT 226
NO_FRAME 227
NO_LOOP_UNROLL 122, 228
NO_RETURN 229
NO_STRING_CONSTR 231, 269
ONCE 232
OPTION 201, 233
SHORT 354
STRING_SEG 235
TRAP_PROC 237, 248
#pragma CREATE_ASM_LISTING 217
#undef 239
%> 105
-! 98
-Ansi 99, 199, 201, 249, 258
-BfaB 100, 205
-C++ 101, 201
-C++c 101
-C++e 101
-C++f 101
-Ca 102
-Cc 94, 103, 213, 214, 216, 218, 220, 221, 222, 231, ... 260, 269
-Ci 105, 201
-Cn 108
-CnCtr 387
-Cni 110, 201, 331
-Cq 113
-CswMaxLF 114
-CswMinLB 116
-CswMinLF 120
-CswMinSLB 118
-Cu 94, 122, 223, 228
-Cv 125
-Cx 126
-D 127, 141
-Ec 128
-Env 57, 130
-F1 . . . . . 34, 131, 204, 260
-F2 . . . . . 34, 131, 204, 260
-F7 131
-Fd 133, 343
-Fh 34, 131, 204
-H 134
-I 135
-La 136
-Ldf 141, 199
-Li 138
-Lic 139
-LicA 140
-Ll 83, 143
-Lm 83, 144
-Lo 83, 146
-Lp 147
-Mb 148, 207, 341
-Ms 148, 207, 341
-N 149
-NoBeep 150
-ObjN 151
-Oc 94, 152, 332
-OdocF 94, 96, 153
-Odocf 201
-Oi 94, 155, 331
-OnP 157
-Ont 159, 470, 471, 472, 473, 474

-Or 94, 161, 331
-Os 162, 201, 263
-Ot 162, 201
-Pe 163
-Pio 165
-Prod 63, 167
-Qvpt 168
-T 169, 205, 342, 461, 462, 466, 553
-V 174
-W1 196
-W2 197
-Wmsg8x3 175
-WmsgFb 56, 175, 176, 179, 181, 183, 184, 185
-WmsgFbi 176
-WmsgFbm 176
-WmsgFi 56, 175, 177, 178, 181, 183, 184, 186
-WmsgFim 178
-WmsgFiv 178
-WmsgFob 180, 183
-WmsgFoi 181, 182, 184, 186
-WmsgFonf 184
-WmsgFonp 181, 183, 184, 185, 186
-WmsgNe 187
-WmsgNi 188
-WmsgNw 189
-WmsgSd 190
-WmsgSe 191
-WmsgSi 192
-WmsgSw 193
-Wpd 94, 194
.c 89
.cpp 89
.cxx 89
.h 89
.hidefaults 57, 73, 74, 86
.ini 40
.lst 479
.o 90
.prm 479
?? 105
??! 105
??(.....105
??) 105
??- 105
??/ 105
??= 105
??> 105
??’ 105
@address 240
@interrupt 554
@tiny 554
__ARCHIMEDES__ 199
__BANKED__ 148, 207
__BIG_ENDIAN__ 200
__BITFIELD_LSBIT_FIRST__ 100, 204, 205, 553
__BITFIELD_LSBYTE_FIRST__ . . 100, 205, 553
__BITFIELD_MSBIT_FIRST__ 100, 204, 205, ... 553
__BITFIELD_MSBYTE_FIRST__ . 100, 204, 205, ... 553
__CHAR_IS_16BIT__ 169, 206
__CHAR_IS_32BIT__ 169, 206
__CHAR_IS_64BIT__ 169, 206
__CHAR_IS_8BIT__ 169, 206
__CHAR_IS_SIGNED__ 169, 206
__CHAR_IS_UNSIGNED__ 169, 206
__CNI__ 110, 201
__cplusplus 101, 201, 280, 282
__DATE__ 199
__DOUBLE_IS_DSP__ 170, 207
__DOUBLE_IS_IEEE32__ 170, 207
__DOUBLE_IS_IEEE64__ 170, 207
__ELF_OBJECT_FILE_FORMAT__ 131, 204
__ENUM_IS_16BIT__ 169, 206
__ENUM_IS_32BIT__ 169, 206
__ENUM_IS_64BIT__ 169, 206
__ENUM_IS_8BIT__ 169, 206
__ENUM_IS_SIGNED__ 169, 206
__ENUM_IS_UNSIGNED__ 169, 206
__FILE__ 199
__FLOAT_IS_DSP__ 170, 207
__FLOAT_IS_IEEE32__ 170, 207
__FLOAT_IS_IEEE64__ 170, 207
__HC08__ 207
__HIWARE__ 199
__HIWARE_OBJECT_FILE_FORMAT__ . . 131, ... 204
__INT_IS_16BIT__ 169, 206
__INT_IS_32BIT__ 169, 206
__INT_IS_64BIT__ 169, 206
__INT_IS_8BIT__ 169, 206
__LINE__ 199
__LITTLE_ENDIAN__ 200
__LONG_DOUBLE_IS_DSP__ ... 170, 207
__LONG_DOUBLE_IS_IEEE32__ . 170, 207
__LONG_DOUBLE_IS_IEEE64__ . 170, 207

__LONG_IS_16BIT__ 169, 206
__LONG_IS_32BIT__ 169, 206
__LONG_IS_64BIT__ 169, 206
__LONG_IS_8BIT__ 169, 206
__LONG_LONG_DOUBLE_DSP__ 170, 207
__LONG_LONG_DOUBLE_IS_IEEE32__ . 170, ... 207
__LONG_LONG_DOUBLE_IS_IEEE64__ . 170, ... 207
__LONG_LONG_IS_16BIT__ 170, 206
__LONG_LONG_IS_32BIT__ 170, 206
__LONG_LONG_IS_64BIT__ 170, 207
__LONG_LONG_IS_8BIT__ 170, 206
__OPTIMIZE_FOR_SIZE__ 162, 201
__OPTIMIZE_FOR_TIME__ 162, 201
__OPTION_ACTIVE__ 201
__PRODUCT_HICROSS__ 200
__PRODUCT_HICROSS_PLUS__ 200
__PRODUCT_SMILE_LINE__ 200
__PTR_SIZE_1__ 207
__PTR_SIZE_2__ 207
__PTR_SIZE_3__ 208
__PTR_SIZE_4__ 208
__PTRDIFF_T_IS_CHAR__ 202, 203
__PTRDIFF_T_IS_INT__ 202, 203
__PTRDIFF_T_IS_LONG__ 203
__PTRDIFF_T_IS_SHORT__ 202, 203
__SHORT_IS_16BIT__ 169, 206
__SHORT_IS_32BIT__ 169, 206
__SHORT_IS_64BIT__ 169, 206
__SHORT_IS_8BIT__ 169, 206
__SIZE_T_IS_UCHAR__ 202, 203
__SIZE_T_IS_UINT__ 202, 203
__SIZE_T_IS_ULONG__ 202, 203
__SIZE_T_IS_USHORT__ 202, 203
__SMALL__ 148, 207
__STDC__ 99, 199, 201
__TIME__ 199
__TRIGRAPHS__ 105, 201
__VERSION__ 200
__VTAB_DELTA_IS_16BIT__ 170, 207
__VTAB_DELTA_IS_32BIT__ 170, 207
__VTAB_DELTA_IS_64BIT__ 170, 207
__VTAB_DELTA_IS_8BIT__ 170, 207
_FAR_CALL 350
_IOFBF 493
_IOLBF 493
_IONBF 493
_OVERLAP 125
_PRESTART 341
_RUNTIME 341

A

abort 480, 486, 496
About Box 54
abs 484, 497
abstract classes 315
acos 483, 497
acosf 484, 497
Adaptation 350
Alignment 344, 347
alloc.c 479
and 105
and_eq 105
ANSI-C 110, 201
Reference Document 239
Standard 239
ANSI-C 239
Array
far 243
asctime 486, 497
asin 483, 497
asinf 484, 497
Assembler 365
assert 482, 498
assert.h 494
Associativity 565
atan 483, 498
atan2 . . . . . .483, 498
atan2f 484, 498
atanf 484, 498
atexi 480
atexit 486, 499
atof 487, 499
atoi 487, 499
atol 487, 500
auto 239

B

BANKED memory model 341
Big Endian 200
Bit Fields 257, 344, 553
bitand 105
bitor 105
Block Nesting 250
Branch Optimization 262, 357

Branch Sequence 263
Branch Tree 264
break 239
bsearch 485, 500
BUFSIZ 493
burner.exe 29

C

C++ 32, 237
C++ Front End 279
C++ Library 543
  I/O library 545
  Memory library 547
  String library 546
Calling Convention 348
calling convention 350
calloc 479, 485, 501
case 239
Cases in Switches 250
catch 281
ceil 483, 501
ceilf 484, 501
char 239, 249
CHAR_BIT 489
CHAR_MAX 489
CHAR_MIN 489
chc08.exe 29
class 281, 292
clearerr 488, 501
clock 486, 502
clock_t 493
CLOCKS_PER_SEC 494
CODE 95
CODE GENERATION 96
Code Size 251
CODE_SECTION 211, 259
CODE_SEG 211, 259, 350
comments 554
Common Source Files 478
comp1 105
compactC++ 32
compactC++ library 543
Compiler
  Advanced Options Dialog 48
  Configuration 40
  Control 50
  Error
    Messages 53
  Error Feedback 55
  Error messages 375
  Graphic Interface 35
  HI-CROSS+ compiler 35
  Include file 89
  Input File 54, 89
  Menu 45
  Menu Bar 39
  Messages 51
  Option 48
  Standard Types Dialog Box 47
  Status Bar 39
  Tool Bar 38
  User Interface 35
compiler generated functions 299
Compiler Pragmas 209
COMPOPTIONS 63, 71, 93
const 103, 239, 265
CONST_SECTION 103, 214, 259
CONST_SEG 214, 259
constructor 297
continue 239
conversion
  by constructor 298
  functions 298
COPYRIGHT 72, 78, 88
cos 483, 502
cosf 484, 502
cosh 483, 502
coshf 484, 502
CREATE_ASM_LISTING 217
ctime 486, 502
ctype 482
ctype.h 495
Current Directory 57, 73
CurrentCommandLine 64

D

DATA_SECTION 218, 259
DATA_SEG 218, 259, 332
Decoder 34
decoder.exe 29
default 239
default arguments 321
Default Directory 59
DEFAULT.ENV 57, 73, 74, 86
default.env 93
DEFAULTDIR 58, 59, 73, 89

DefaultDir 59
define 127
defined 239
delete 281, 321
destructor 297
difftime 486, 503
DIG 489
Directive
  #define 141, 239
  #elif 239
  #else 239
  #endif 239
  #error 239
  #if 239
  #ifdef 239
  #ifndef 239
  #include 138, 239
  #line 239
  #pragma 239
  #undef 239
  Preprocessor 239
div 484, 503
div_t 493
Division 249, 346
do 239
DOS 98
double 239

E

EBNF 561
Editor 63
Editor_Exe 61, 63
Editor_Name 61, 63
Editor_Opts 61, 64
EditorCommandLine 67
EditorDDEClientName 67
EditorDDEServiceName 68
EditorDDETopicName 67
EditorType 67
EDOM 489
EDOUT 90
ELF/DWARF 33, 34, 48
else 239
Endian 200
Entry Code 351
enum 239
ENVIRONMENT 74
Environment
  COMPOPTIONS 71, 93
  COPYRIGHT 72, 78, 88
  DEFAULTDIR 58, 59, 73, 89
  ENVIRONMENT 57, 74
  ERRORFILE 75, 90
  File 57
  GENPATH 77, 79, 80, 89, 135
  HICOMPOPTIONS 71
  HIENVIRONMENT 74
  HIPATH 77, 80
  HITEXTFAMILY 81
  HITEXTKIND 82
  HITEXTSIZE 84
  HITEXTSTYLE 85
  INCLUDETIME 72, 78, 88
  LIBPATH 77, 79, 87, 89, 90, 135
  LIBRARYPATH 79, 89, 90, 135
  OBJPATH 80, 90, 151
  TEXTFAMILY 81
  TEXTKIND 82
  TEXTPATH . . 83, 136, 141, 143, 144, 146
  TEXTSIZE 84
  TEXTSTYLE 85
  TMP 86
  USELIBPATH 79, 87
  USERNAME 72, 78, 88
  Variable 57, 70
Environment Variable 209
EOF 493
EPROM 242
EPSILON 489
ERANGE 489
errno 489
errno.h 489
Error
  Handling 482
  Listing 90
  Messages 53
Error Format
  Microsoft 178
  Verbose 178
Error Listing 90
Error messages 375
ERRORFILE 75, 90
Escape Sequences 566
exit 480, 486, 503
Exit Code 352
EXIT_FAILURE 493
EXIT_SUCCESS 493

exp 483, 503
expf 484, 503
Explorer 58
Expression Recursion 250
Extended Backus-Naur Form, see EBNF
extern 239
extern "C" 281

F

fabs 483, 504
fabsf 484, 504
FAR 350, 354
far 240, 342, 350
  Arrays 243
  Keyword 242
Far Calling Convention 349
fclose 487, 504
feof 487, 504
ferror 487, 504
fflush 487, 505
fgetc 488, 505
fgetpos 488, 505
fgets 488, 506
FILE 492
File
  Environment 57
  Include 89
  Object 90
  Source 89
File Manager 58
File Names 251
FILENAME_MAX 493
float 239
float.h 489
Floating Point 343
floor 483, 506
floorf 484, 506
FLT_RADIX 489
FLT_ROUNDS 489
fmod 483, 506
fmodf 484
fopen 487, 507
FOPEN_MAX 493
for 239
fpos_t 492
fprintf 488, 508
fputc 488, 508
fputs 488, 508
Frame Pointer 351
Frame, see Stack Frame
fread 488, 508
free 479, 485, 508
freopen 487, 509
frexp 483, 509
frexpf 484, 509
friend 281, 296
Front End 239
fscanf 488, 509
fseek 488, 509
fsetpos 488, 510
ftell 488, 510
Function Pointer 351
fwrite 488, 511

G

GENPATH 77, 79, 80, 89, 135
getc 488, 511
getchar 488, 511
getenv 486, 511
gets 488, 511
gmtime 486, 512
goto 239, 250
Group 59
GUI Graphical User Interface 35

H

HALT 479, 480
Hardware 350
heap.c 479
HI-CROSS+ 31, 34, 35, 203, 342, 343
HICOMPOPTIONS 71
HIENVIRONMENT 74
High Address Part 373
HIPATH 77
HITEXTFAMILY 81
HITEXTKIND 82
HITEXTSIZE 84
HITEXTSTYLE 85
hitools.exe 29
HIWAVE 34
hiwave.exe 29
HOST 95, 96
HUGE_VAL 491

I

I/O library 545
I/O Registers 242
IEEE 343
if 239
Implementation Restriction 250
Include File Recursion 250
Include Files 89, 250
INCLUDETIME 72, 78, 88
inheritance 306
INLINE 155, 221
inline 155, 281, 285, 331
Inline Assembler, see Assembler
INPUT 95, 96
int 239
INT_MAX 490
INT_MIN 490
Intel 200
Internal ID's 250
Interrupt 240, 248, 353, 554
    keyword 248
    vector 248
interrupt 237, 342
INTO_ROM 103, 222
IPATH 80
isalnum 485, 512
isalpha 485, 512
iscntrl 485, 512
isdigit 485, 512
isgraph 485, 512
islower 485, 512
isprint 485, 512
ispunct 485, 512
isspace 485, 512, 513
isupper 485, 512, 513
isxdigit 485, 512, 513

J

jmp_buf 492
Jump Table 264

K

Keywords 342

L

Labels 250
labs 484, 513
LANGUAGE 96, 108
Lazy Instruction Selection 356
lconv 490
ldexp 483, 513
ldexpf 484, 513
ldiv 484, 513
ldiv_t 493
Lexical Tokens 250
libmaker.exe 29
LIBPATH 77, 79, 87, 89, 90, 135
Library Files 477, 478
Library Structure 482
LIBRARYPATH 79, 89, 90, 135
Limits
    Translation 250
limits.h 489
Line Continuation 69
Linker 34
linker.exe 29
Little Endian 200
locale.h 490
localeconv 487, 514
Locales 482
localtime 486, 514
log 483, 514
log10 . . . . . .483, 514
log10f 484, 514
logf 484, 514
long 239
LONG_MAX 490
LONG_MIN 490
longjmp 486, 515
LOOP_UNROLL 122, 223

M

Macro 127
    Predefined 199
Macro Expansion 250
Macros 250
maker.exe 29
malloc 479, 485, 515
MANT_DIG 489
math.h 491, 528
MAX 489
MAX_10_EXP 489

MAX_EXP 489
MB_LEN_MAX 489, 493
mblen 480, 485, 515
mbstowcs 480, 486, 515
mbtowc 480, 486, 516
MCUTOOLS.INI 41, 58, 73
Member Access Control 294
member functions 293
memchr 483, 516
memcmp 483, 516
memcpy 483, 516
memmove 483, 516
Memory library 547
Memory Models 341
memset 483, 517
MESSAGE 96, 224
MESSAGES 95
Messages Settings 51
Microsoft 178
MIN 489
MIN_10_EXP 489
MIN_EXP 489
mktime 486, 517
modf 484, 517
modff 484, 517
Modulus 249, 346
MSB 373
MS-DOS 98
multiple inheritance 303, 306

N

name encoding 326
NEAR 350, 354
near 240, 342, 350
Near Calling Convention 348
new 281, 321
NO_ENTRY 225, 367
NO_EXIT 226
NO_FRAME 227
NO_LOOP_UNROLL 122, 228
NO_RETURN 229
NO_STRING_CONSTR 231, 269
NON_BANKED 341
not 105
not_eq 105
NULL 492
Numbers 250

O

Object
File 90
OBJPATH 80, 90, 151
offsetof 492
ONCE 232
operator 281
. 239
. 239
defined 239
operator delete 321
operator new 321
OPTIMIZATION 95, 96, 110
Optimization
Branches 262, 357
Lazy Instruction Selection 356
Register Tracing 357
Shift optimizations 262
Strength Reduction 262, 356
Time vs. Size 162, 357
Tree Rewriting 263
OPTION 233
Option
CODE 95
CODE GENERATION 96
HOST 95, 96
INPUT 95, 96
LANGUAGE 96, 108
LANGUAGE 95
MESSAGE 96
MESSAGES 95
OPTIMIZATION 95, 96, 110
OUTPUT 95, 96
TARGET 95
VARIOUS 95, 96
Options 59, 66
or 105
or_eq 105
OUTPUT 95, 96
overloading 321

P

Parsing Recursion 250
Path 59
Path List 68
perror 482, 517
Pointer
Compatibility 247

far 242
pow 484, 518
powf 484, 518
Precedence 565
Predefined Macro 199
Preprocessor
Directives 239
printf 487, 488, 518
printf.c 480
private 281, 294
Procedure
Return Value 348
Stack Frame 351
project.ini 63, 71, 93
protected 281, 294
ptrdiff_t 201, 492
public 281, 294
pure virtual functions 315
putc 488, 518
putchar 488, 518
puts 488, 519
PVCS 87

Q

qsort 485, 519

R

raise 486, 520
rand 484, 520
RAND_MAX 493
realloc 479, 485, 520
RecentCommandLine 64
Recursive comments 554
Register
Tracing 357
register 239
Register Tracing 357
remove 488, 520
rename 488, 521
Restriction
Implementation 250
return 239
Return Value 348
rewind 488, 521
ROM 265
rom 240, 246, 342
ROM_VAR 94, 103, 260

S

SaveAppearance 60
SaveEditor 60
SaveOnExit 59
SaveOptions 60
scanf 488, 521
SCHAR_MAX 489
SCHAR_MIN 489
SEEK_CUR 493
SEEK_END 493
SEEK_SET 493
Segment 353
SHORT 259, 354
Segmentation 258
setbuf 488, 521
setjmp 486, 522
setjmp.h 492
setlocale 487, 522
setvbuf 488, 523
Shift optimizations 262
SHORT 332, 354
short 239
SHORT Segments 354
ShowTipOfDay 66
SHRT_MAX 489
SHRT_MIN 489
sig_atomic_t 492
SIG_DFL 492
SIG_ERR 492
SIG_IGN 492
SIGABRT 492
SIGFPE 492
SIGILL 492
SIGINT 492
signal 486, 523
signal.c 479
signal.h 492
Signals 479
signed 239
SIGSEGV 492
SIGTERM 492
sin 484, 524
sinf 484, 524
single inheritance 302
sinh 484, 524
sinhf 484
Size
Type 342
size_t 201, 492

sizeof .239
SMALL memory model .341
Smart Control .50
Smile~Line .31, 34
Source File .89
special member functions .297
sprintf .487, 524
sqrt .484, 528
sqrtf .484, 528
srand .484, 528
sscanf .487, 528
Stack
Frame .351
Stack Frames .351
Stack usage .351
Standard Types .47
STARTUP .341
startup .63
Startup Files .478
static .239
static members .293
StatusbarEnabled .65
stdarg.h .494
stddef.h .492
stderr .493
stdin .493
stdio.h .492
stdlib. .480
stdlib.c .480
stdlib.h .493, 528, 541
stdout .493
strcat .482, 531
strchr .483, 531
strcmp .483, 532
strcoll .487, 532
strcpy .482, 532
strcspn .483, 532
Strength Reduction .262, 356
strerror .482, 533
strftime .486, 533
String Concatenation .250
String Length .250
String library .546
string.h .494
STRING_SECTION .235
STRING_SEG .235
Strings .242
strlen .482, 534
strncat .483, 534
strncmp .483, 535
strncpy .482, 535
strpbrk .483, 535
strrchr .483, 535
strspn .483, 535
strstr .483, 536
strtod .487, 536
strtok .483, 536
strtol .487, 537
strtoul .487, 537
struct .239
Structs & Unions .250
strxfrm .487, 538
switch .239
system .486, 538

T

tan .484, 538
tanf .484, 538
tanh .484, 538
tanhf .484, 538
TARGET .95
template .281
templates .312
TEXTFAMILY .81
TEXTKIND .82
TEXTPATH 83, 136, 141, 143, 144, 146, 147
TEXTSIZE .84
TEXTSTYLE .85
this .281
throw .281
time .486, 539
time.h .493
time_t .493
Tip of the Day .35
TipFilePos .66
TMP .86
TMP_MAX .493
tmpfile .488, 539
tmpnam .488, 539
tolower .485, 539
ToolbarEnabled .65
toupper .485, 540
Tracing, see Register Tracing
Translation Limits .250
TRAP_PROC .237, 248, 352, 554
try .281
Type

Floating Point 343
Size 342
Type Declarations 250
typedef 239

U

UCHAR_MAX 489
UINT_MAX 490
ULONG_MAX 490
ungetc 488, 540
uni 240, 247, 342
union 239
UNIX 58
unsigned 239
USELIBPATH 79, 87
USERNAME 72, 78, 88
USHRT_MAX 490

V

va_arg 486, 540
va_end 486, 540
va_start 486, 540
Variable
Environment 57
VARIOUS 95, 96
VECTOR 248
vfprintf 488, 541
virtual 281, 289
virtual base classes 306
virtual functions 299
void 239
volatile 239, 257
vprintf 488, 541
vsprintf 487, 541

W

wchar_t 492
wcstombs 480, 486, 541
wctomb 480, 486, 541
while 239
Win32s 37
WindowFont 66
WindowPos 65
Windows 58
WinEdit 58, 76

X

xor 105
xor_eq 105

Z

Zero Page 354