

RadiSys.

Using Ultra C/C++

Version 2.6

Copyright and publication information

This manual reflects version 2.6 of Ultra C/C+++.
Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Corporation.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

October 2002
Copyright ©2003 by RadiSys Corporation.
All rights reserved.

EPC and RadiSys are registered trademarks of RadiSys Corporation. ASM, Brahma, DAI, DAQ, MultiPro, SAIB, Spirit, and ValuePro are trademarks of RadiSys Corporation.

DAVID, MAUI, OS-9, OS-9000, and SoftStax are registered trademarks of RadiSys Corporation. FasTrak, Hawk, and UpLink are trademarks of RadiSys Corporation.

† All other trademarks, registered trademarks, service marks, and trade names are the property of their respective owners.

Table of Contents

Chapter 1: Overview

13

-
- 14 Treatment of Processor-Specific Information
 - 14 Microware Version 3.2 K&R C Compiler
 - 14 Treatment of Code Comments
 - 16 Compiler Features
 - 16 ANSI- and ISO-Compliance
 - 17 Optimizing
 - 17 Integration with Existing Microware Products
 - 19 Compiler Components
 - 23 Executive
 - 27 I-Code Linker
 - 31 Object Code Linker
 - 34 Source Code Compatibility

Chapter 2: Compiling

37

-
- 38 main() Function Parameters
 - 39 Code Size
 - 39 Defaults
 - 39 Methods for Reducing Compiled Code Size
 - 42 Code Speed
 - 42 Optimization Options
 - 44 Time and Space Control
 - 45 Maximizing Speed
 - 46 Minimizing Space
 - 46 Using Time/Space Ratios
 - 47 Floating Point Math
 - 47 Hardware Emulation
 - 47 Software Emulation

48	Assembly Language in C Source Files
50	External <code>_asm()</code> Pseudo Function
53	External <code>_asm()</code> Statements
54	Embedded <code>_asm()</code> Statements
71	Using the Size Parameter
72	Suggestions Related to Using <code>_asm()</code> Statements
81	Error Messages
84	Stack Checking
84	Non-Program Modules
86	Multi-Threading
86	Libraries
88	CSL and Multi-Threaded Applications
89	Keywords
89	volatile
90	const
91	remote
92	C++ Features and Restrictions
93	Object Size and Alignment
94	Compatibility with the Microware K&R C Compiler
95	Running Compiler Makefiles
96	Differences Between Compatibility Source Mode and the Microware K&R C Compiler
96	Using Remote as a Storage Class
97	C Keywords and Struct Member Names
97	Error Checking
97	Name Clashing
97	The Executive
98	Using csl
98	Multiple Copies of Some Library Functions
100	Using Libraries
101	Using Assembly Language
101	Using <code>deasm</code>
102	Standard I/O and Microware K&R C Compiler ROFs

-
- 106 The Executive (xcc) Option Modes
 - 107 Environment Variables
 - 107 CC: Change the Executive Option Mode
 - 108 CDEF: Select Directories for Standard #Include Files
 - 108 CLIB: Select Directories for Standard I-Code and Object Code Library Files
 - 109 MWOS: Set the Root for the Standard MWOS File Structure
 - 110 TMPDIR: Select a Device or Directory for Temporary Files
 - 111 Include File Search Path Algorithm
 - 112 Library File Search Path Algorithm
 - 114 Predefined Macro Names for the Preprocessor
 - 117 Compiler Phase Codes
 - 119 Option Modes
 - 119 ucc Option Mode
 - 120 c89 Option Mode
 - 123 compat Option Mode
 - 125 Library Naming Conventions
 - 126 Command Line Options

-
- 158 Single Source File Program
 - 159 Single Source File Program I-Code Linked with I-Code Libraries
 - 160 Multiple Source File Program I-Code Linked
 - 161 Multiple Source File Program Object Code Linked
 - 162 Multiple Source File Program I-Code Linked with Makefile
 - 166 Multiple Source File Program Object Code Linked with Makefile
 - 167 Multiple Source File Program I-Code Linked into Segments with Makefile
 - 171 Multiple Source File Non-Program I-Code Linked with Makefile
 - 173 Multiple Source File Object Library Creation with Makefile
 - 175 Multiple Source File I-Code Library Creation with Makefile

Chapter 5: Compiler Phase Options

179

180	Passing Options
183	Front End
200	I-Code Linker
202	I-Code Optimizer
208	Back End
210	Assembly Optimizer
212	Assembler
215	Prelinker
217	Object Code Linker

Chapter 6: Assembler and Object Code Linker Overview

223

224	Assembler
226	Linker
229	The Assembly Language Program Development Process
230	Relocatable Program Sections
231	Program Section Declarations: psect and vsect
231	psects
232	vsects
235	Example Program Layout for a Motorola OS-9 for 68K Program
237	Location Counters
238	Relocatable Object File Format
240	ROF Edition Number 9 Format
240	Header Section
243	External Definition Section
244	Object Code Section
245	Initialized Data Section
245	Remote Initialized Data Section
245	Debug Information Section
246	External Reference Definition Section
247	Local Reference Section

250	ROF Edition Number 15 Format
250	Header Section
254	External Definition Section
255	Object Code Section
255	Initialized Data Section
255	Remote Initialized Data Section
256	Constant Data Section
256	Remote Constant Data Section
256	Debug Information Section
256	Externally Referenced Symbol Data Section
257	Expression Tree Data Section
261	Reference Data Section

Chapter 7: Assembler

263

264	Running the Assembler
266	Symbolic Names
267	Evaluating Expressions
268	Expression Operands
270	Expression Operators
272	Expressions Involving External Symbols
273	Input File Format
273	Label Fields
276	The Operation Field
277	Operand Field
277	Comment Field
278	Assembly Listing Format
279	Macros
280	Structure
282	Arguments
284	Automatic Internal Labels
285	Directive Statements
310	Pseudo-Instructions

Chapter 8: Prelinker

325

- 326 Automatic Instantiation
- 329 Prelinker
- 330 Prelinker Execution
 - 330 Description and Example
- 334 Creating C++ Libraries Containing Templates
 - 334 Terms and Definitions
 - 335 Creating Libraries which Reference Templated Entities
- 339 Building C++ Libraries Referencing Template Entities
- 340 Structuring and Building Template Libraries

Chapter 9: Object Code Linker

343

- 344 Purpose
 - 345 psects
- 346 Usage
- 348 Text Output
- 350 Execution
 - 351 First Phase
- 355 Library Files
 - 355 Libgen
 - 355 Merged Libraries
- 357 Library Format Created by libgen
 - 357 Library Header
 - 359 Global Definition Hash Table
 - 359 Global Definition Section
 - 360 String Table
 - 360 psect Section
 - 363 Internal Reference Section
 - 364 External Reference Section
- 365 Linker Defined Symbols
- 370 Module Header Override Symbols
- 372 Linking Code for Non-OS-9 Systems

384	Constant Propagation
385	Constant Folding
386	Loop Rotation
387	Variable Lifetimes
388	Register Coloring and Coalescing
389	Common Subexpressions
390	Pointer Tracking with CSE
391	Useless Code Elimination
393	Useless Copy Elimination
395	Useless Pointer Elimination
397	Assignment Translation
398	Code Motion and Combining
398	Common Successor Code
399	Common Predecessor Code
399	Common Tail Code
401	Loop Optimizations
401	Initial Loop Condition Testing
402	Invariant Hoisting
403	Strength Reduction
403	Loop Unrolling
405	Constant Sharing
406	Function Inlining
408	Span Dependent Optimizations
409	Assembly Level Optimizations
409	Merging Common Tails
410	Reducing Branch Complexity
411	Location Tracking
411	Offsetting Stack Changes
411	Pipeline Scheduling

Chapter 12: Language Features

413

414	C Language Features
414	Translation
415	Environment
416	Identifiers
417	Characters
418	Integers
418	Floating Point
419	Structures, Unions, Enumerations, and Bit-Fields
419	Qualifiers
419	Declarators
419	Statements
420	Preprocessing Directives
421	C++ Language Features
421	Dialect Accepted
425	Anachronisms Accepted
426	Extensions Accepted
428	Template Instantiation
430	Instantiation Modes
431	Instantiation #pragma Directives
433	Implicit Inclusion
434	Compiling with Exceptions Disabled
441	Wide Character Strings

Appendix A: Messages

443

444	Executive
450	Front End
491	I-Code Linker
498	I-Code Optimizer
503	Back End
510	Assembly Optimizer
513	Assembler

- 522 Prelinker
- 524 Object Code Linker

Appendix B: Migrating to Ultra C++ version 2.1 **535**

- 537 New Language Features
- 542 Language Features in the C++ Standard but Not Accepted
- 544 Headers
- 545 Namespaces
- 547 Operators new and delete
- 549 New Template Features
- 551 The Standard C++ Library
- 554 Prelinker
- 554 Frequently Asked Questions about Migrating to Ultra C++ v2.1
- 566 Some Observations

Index **567**

Chapter 1: Overview

Ultra C/C++ is a high-performance C/C++ compiler that produces fast, efficient code for real-time applications. It conforms to the American National Standards Institute (ANSI) and International Standards Organization (ISO) C standards. It also tracks the C++ ANSI/ISO draft standard. With minimal changes, applications written with Ultra C/C++ on OS-9, as well as on other platforms that comply with the ANSI/ISO C standard, may be compiled. ANSI/ISO C-conforming code developed under other ANSI/ISO C platforms is easily ported to Ultra C/C++.

The following sections are included in this chapter:

- **Treatment of Processor-Specific Information**
- **Compiler Features**
- **Compiler Components**
- **Source Code Compatibility**



Treatment of Processor-Specific Information

Many examples used in this manual reflect the Motorola 68K family of processors. Ultra C/C++ information specific to a particular processor is identified in the *Ultra C/C++ Processor Guide*.

Microware Version 3.2 K&R C Compiler

The Microware version 3.2 K&R C Compiler (pre-Ultra C compiler) is a superset of the first edition of Kernighan and Ritchie (K&R). To port existing code developed under Microware version 3.2 of the K & R C (non-ANSI/ISO-conformant) compiler for 68K or Microware Version 1.3 of the K & R C (non-ANSI/ISO-conformant) compiler, refer to [Chapter 2](#).



Note

Microware version 3.2 of the K&R C compiler (pre-Ultra C compiler) is referred to in the remainder of this manual as the Microware K&R C compiler.

Treatment of Code Comments

There are two categories of code comments used in code examples in this manual: actual code comments and reader information comments. Comments, whether code or reader, compile without error; they are formatted as follows:

Sample C Code

```

/* Default stack handler function */

/* typedef to the 1 byte unit so pointer arithmetic
   is easy */
typedef unsigned char byte;

extern unsigned long _;      /* max stack used so far */
extern byte *_sttop,        /* stack pointer at start */
           *_stbot,        /* deepest stack depth so far */
           *_mtop,         /* overflow point of stack */
           *__asm_stack_ptr();
                           /* current stack pointer */
extern int _stklimit;      /* space between stack_ptr and
                           /* worst(_stbot) */
extern void _stack_overflow();

```

Sample Assembly Code

```

ident

myfun:                                * myfun() {
    move.l #12,mystruct+4(a6) * mystruct.b = 3*4;
    move.b #99,mystruct+8(a6) * mystruct.c = 'c';
    move.l #10,d0                    * return (sizeof(thing));
    rts                               * }

```

In cases where reader comments in actual code cause compiler errors and where annotations were made manually to generated output, comments are formatted as follows:

```

move.l  a(a6),d7
add.l   d7,d7      //(a * 2)
move.l  #0x100,d0  //immediate value
add.l   a(a6),d0   //global variable
add.l   -4(a5),d0  //stack variable
add.l   d1,d0      //register variable
add.l   d7,d0      //arithmetic expression

```

Compiler Features

- Complies with ANSI and ISO C standards
- C front 2.1-, 3.0-, and ARM-compliant, tracking ANSI/ISO C++ draft standard
- Generates highly-optimized code
- Integrates with existing Microware operating systems and run-time environments

ANSI- and ISO-Compliance

The compiler is compliant with the American National Standard Institute (ANSI) and International Standards Organization (ISO) American National Standard for Programming Language — C (ANSI/ISO 9899-1990) standards. It is also tracking the ANSI/ISO C++ draft standard.



For More Information

Refer to **Chapter 12: Language Features** for more information.

By complying with the ANSI/ISO standards, programs are easily ported between compilers conforming to the standards.

Optimizing

Optimizing increases execution speed and reduces the size of the final object code. The Ultra C/C++ compiler incorporates optimization principles derived from academic and industrial research and algorithms published in recent computer science conference proceedings. Individual compiler phases perform optimizations, each with its own emphasis. Each phase eliminates non-optimal constructions from the previous phase resulting in a highly-optimized executable.



Note

Although current optimization principles and techniques are inherent in the compiler, an executable compiled with Ultra C/C++ may be larger than an executable for the same program compiled with the Microware K&R C compiler. This is usually not an optimization problem; some functions in the Ultra C/C++ library are larger and more complex than those in the Microware K&R C compiler.

Integration with Existing Microware Products

The compiler runs on both OS-9 for 68K (Version 2.4 or greater) and OS-9 (Version 2.0 or greater) without modification. The compiler also compiles source files written with the Microware K&R C compiler. However, the K&R C compiler allowed some illegal source code (such as code using C-reserved words as structure member names to compile). Ultra C/C++ does not compile such code. Also, code written to depend on knowledge of unspecified behavior of the old compiler (such as what registers register variables are kept in) does not work. We cannot guarantee that behavior the ANSI/ISO C/C++ standards say is indeterminate, unspecified, or undefined will be consistent across changes in target or compilation options or from one version of the compiler to the next.



Note

For more information regarding compatibility, refer to the *Ultra C/C++ Processor Guide*.

Compiler Components

The compiler comprises nine logical components: the executive and eight phases, some of which may not exist or which remain as separate components or programs. **Table 1-1** identifies compiler components in the order of execution. Some phases are not always required functions of compiling and may not be executed by the executive component. Required phases are dependent upon the compilation option specified. Components not identified in the table include libraries, utilities or include files.



For More Information

Chapter 3 identifies the options for each executive option mode.

Table 1-1 Compiler Components

Components	Description
Executive	<p>User interface and control program for all other elements. The executive is tri-modal; it can operate in three separate, mutually exclusive option modes. Each option mode executes the phases necessary to process input files to a specified point.</p> <p>The executive is described in detail in Chapter 3.</p>
Front End	<p>The front end phase preprocesses and translates the source file into intermediate code (I-code). The I-code produced by the front end is a machine- and source language-independent binary representation of the source file. By using I-code, the language front ends and target back ends remain fully independent, enabling enhancements to the compiler system without writing a new compiler for each language or target processor.</p>
I-Code Linker	<p>The I-code linker phase merges multiple I-code files into one I-code file. The I-code linker allows partial linking of a program, as well as full linking including I-code libraries. The I-code linker also builds I-code libraries.</p>

Table 1-1 Compiler Components (continued)

Components	Description
I-Code Optimizer	<p>The I-code optimizer phase performs the language- and machine-independent optimizations. The I-code optimizer optimizes on the local (within straight line code) and global (within a whole function) levels.</p> <p>Because the I-code linker can provide an I-code representation of the entire program (including libraries), the I-code optimizer realizes all functions and data within the program and optimizes it as a whole. This is a distinct advantage over compilers that optimize only one function or file at a time.</p> <p>Optimizations performed by the compiler are described in Chapter 5: Compiler Phase Options.</p>
Back End	<p>The back-end phase translates an I-code file into the target assembly language and performs machine-dependent optimizations. The back end performs the following functions:</p> <ul style="list-style-type: none">• Lays out the data area• Selects code for generation according to time and space considerations• Assigns registers based on greatest need• Generates target assembly language code

Table 1-1 Compiler Components (continued)

Components	Description
Assembly Optimizer	The assembly optimizer phase performs machine-specific optimizations. Because the I-code optimizer and the back end perform several optimizations, the assembly optimizer optimizes sequences of code that the back end could not (for example, merging duplicated sections of code into a common section).
Assembler	<p>The assembler phase translates the assembly file into a Relocatable Object File (ROF). One assembler may target all members of the processor family (for example, 68K, 80x86, PowerPC). The assembler performs span-dependent optimization that enables production of code with optimally-sized code references. Span-dependent optimization is most effective when the application is I-code linked since I-code linking reduces the number of external references. This greatly increases the speed and decreases the size of large applications.</p> <p>Information on span-dependent optimization is provided in Chapter 5.</p>

Table 1-1 Compiler Components (continued)

Components	Description
Prelinker	<p>The prelinker phase performs automatic instantiation of templates when linking object- or I-code files. The Prelinker iteratively instantiates templates and recompiles sources until all referenced instantiations are resolved.</p> <p>The Prelinker is described in Chapter 8.</p>
Object Code Linker	<p>The object code linker phase links ROFs with libraries to produce an OS-9 module. Very fast linking is possible when linking with libraries generated by the <code>libgen</code> utility.</p> <p>The <code>libgen</code> utility is described in Chapter 9.</p>

Executive

Use the executive `xcc` to compile source code. The syntax for `xcc` is shown below:

```
xcc [-mode=<mode>] [<opts>] <files> [<opts>]
```

`-mode` specifies the executive option mode for a specific compilation. `xcc` option modes are shown below:

- `compat`
- `c89`
- `ucc`

The executive (`xcc`) operates in separate, mutually exclusive option modes defined in [Table 1-2](#).

Table 1-2 Executive Option Modes

Mode	Description
<code>compat</code>	Compatible with the Microware K&R C compiler executive
<code>c89</code>	Similar to the POSIX 1003.2 compiler
<code>ucc</code>	Default mode developed for Ultra C/C++. This mode enables greater control of the phases.

The `CC` environment variable may be used to set the executive option mode for future compiles. The `CC` environment variable can have the same values as the `-mode` executive option.



For More Information

Executive and `CC` option modes are defined in greater detail in [Chapter 3](#).

Options specified on the `xcc` command line are specific to the selected executive option mode. For example, in the following command line the `-j` option causes the compiler to include I-code versions of the standard library specified on the I-code link line. This occurs because the executive option mode is unspecified, thereby defaulting to `ucc`:

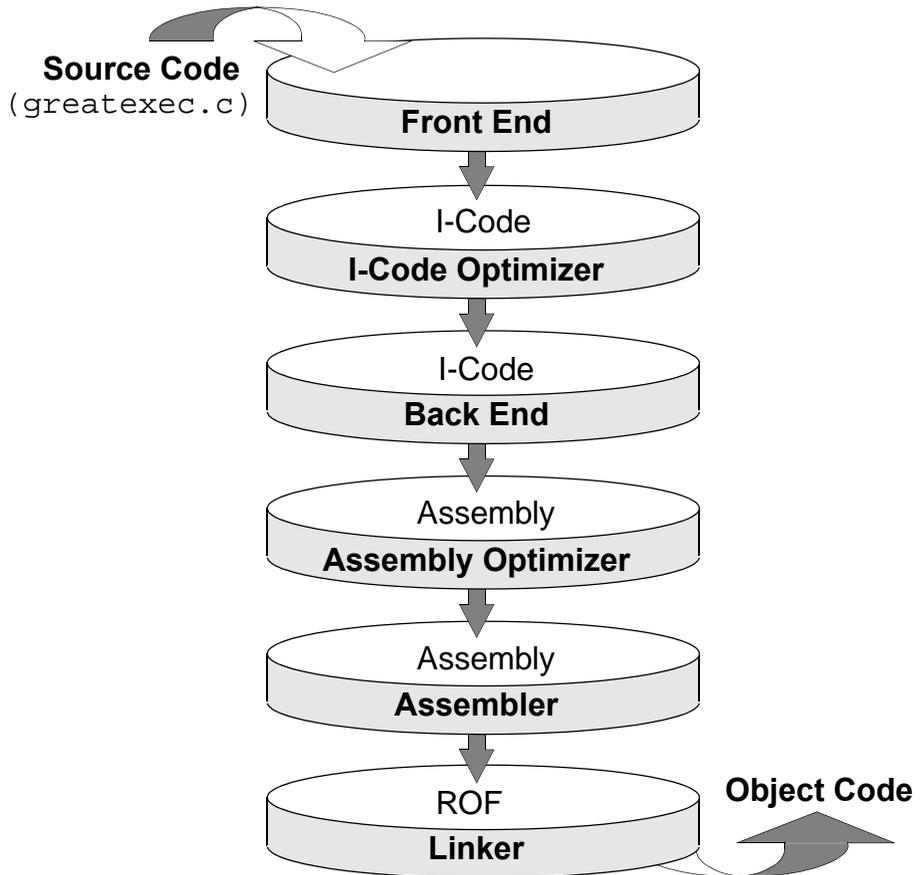
```
xcc greatexec.c -j
```

In the next example, the `-j` option prevents the linker from creating a jump table because the executive option mode specified is `compat`:

```
xcc -mode=compat greatexec.c -j
```

A single C source file may be compiled as shown in **Figure 1-1**.

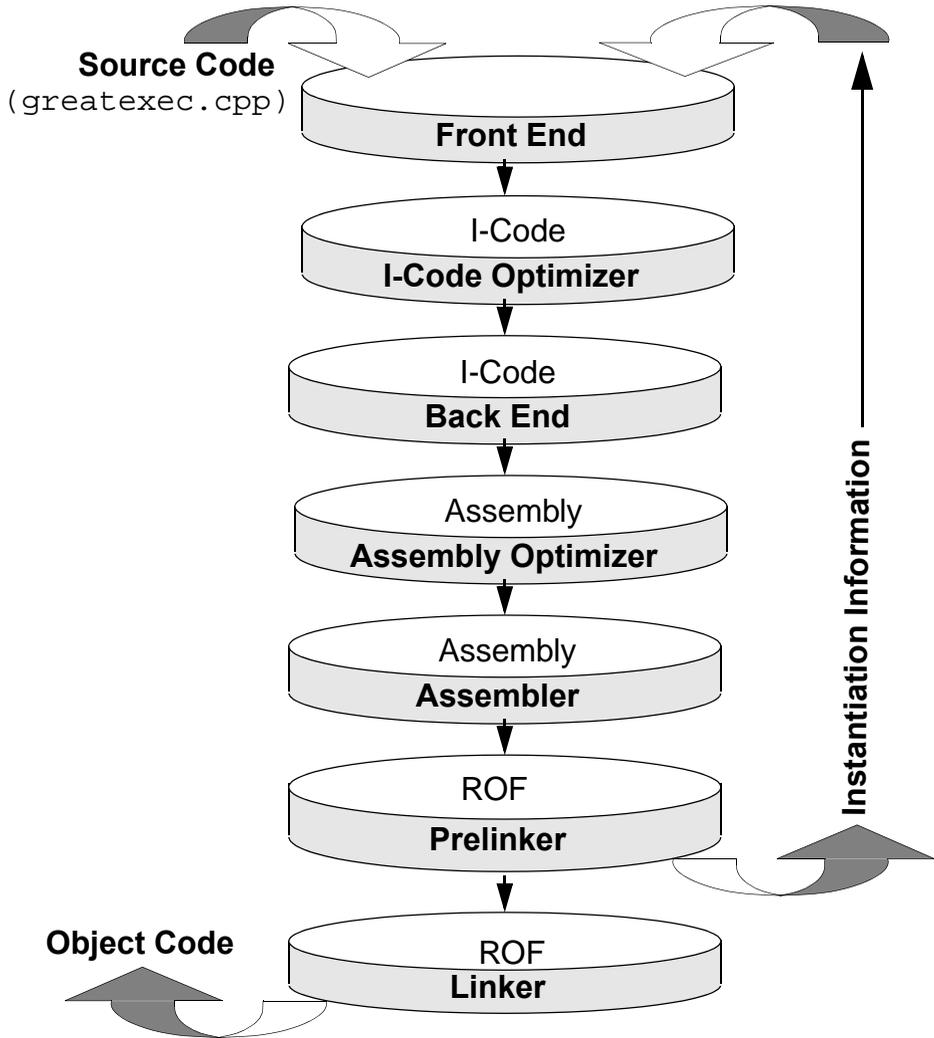
Figure 1-1 Compiling a Single C Source File



When compiling a single C source file, the source passes through each phase of the compiler and is compiled into an object code module.

A single C++ source file may be compiled as illustrated in **Figure 1-2**.

Figure 1-2 Compiling a Single C++ Source File



Two methods of compiling and linking programs are provided:

- Object code linking
- I-code linking

If more than one source file requires compilation and linking, determine whether to link the files before or after optimization.

I-code linking provides the opportunity for the greatest optimization since the entire application can be optimized as a unit. Generally, I-code linking is performed at or near production compile time, since the opportunities for optimization provided by I-code linking can result in longer compilation time. Object code linking is recommended when compilation time is more important than level of optimization, such as during the development phase.

I-Code Linker

When using the I-code link method, the compiler performs the following steps:

-
- Step 1. For C code, compiles each source file to its I-code file.
 - Step 2. For C++ code, prelinks the I-code files, the standard library, and any specified I-code libraries.
 - Step 3. Links the I-code files.
 - Step 4. Optionally links I-code files with specified I-code libraries.
 - Step 5. Processes the file containing the linked I-code files to an OS-9 module.
-

Use of the I-code link method to compile source files enables production of higher quality code than the object code link method. Each file is linked and optimized every time source files are compiled. The compiler realizes the content of the entire application after linking, thereby enabling greater optimization.

To use the I-code link method, enter a command line similar to the following code:

```
xcc main.c greatexec.c compileit.c -f=prog
```

`-f=prog` is an option passed to the compiler. `main.c`, `greatexec.c`, and `compileit.c` are source code files linked in the I-code linker phase.

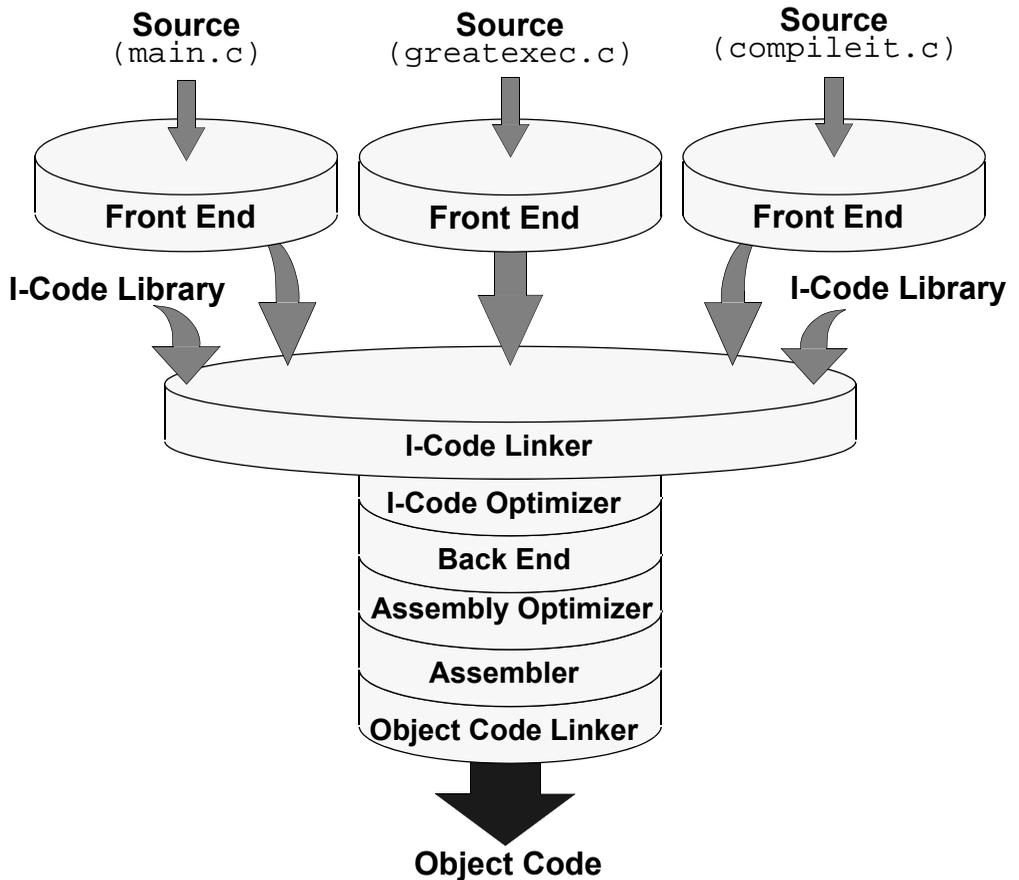


Note

Any partial representation (minimally, one file and one library) or an entire program (all files and libraries) may be I-code linked. The I-code linker resolves as much as it can, allowing unresolved symbols to pass through for resolution at the object link phase.

I-code link processing of C source files is illustrated in the following figure.

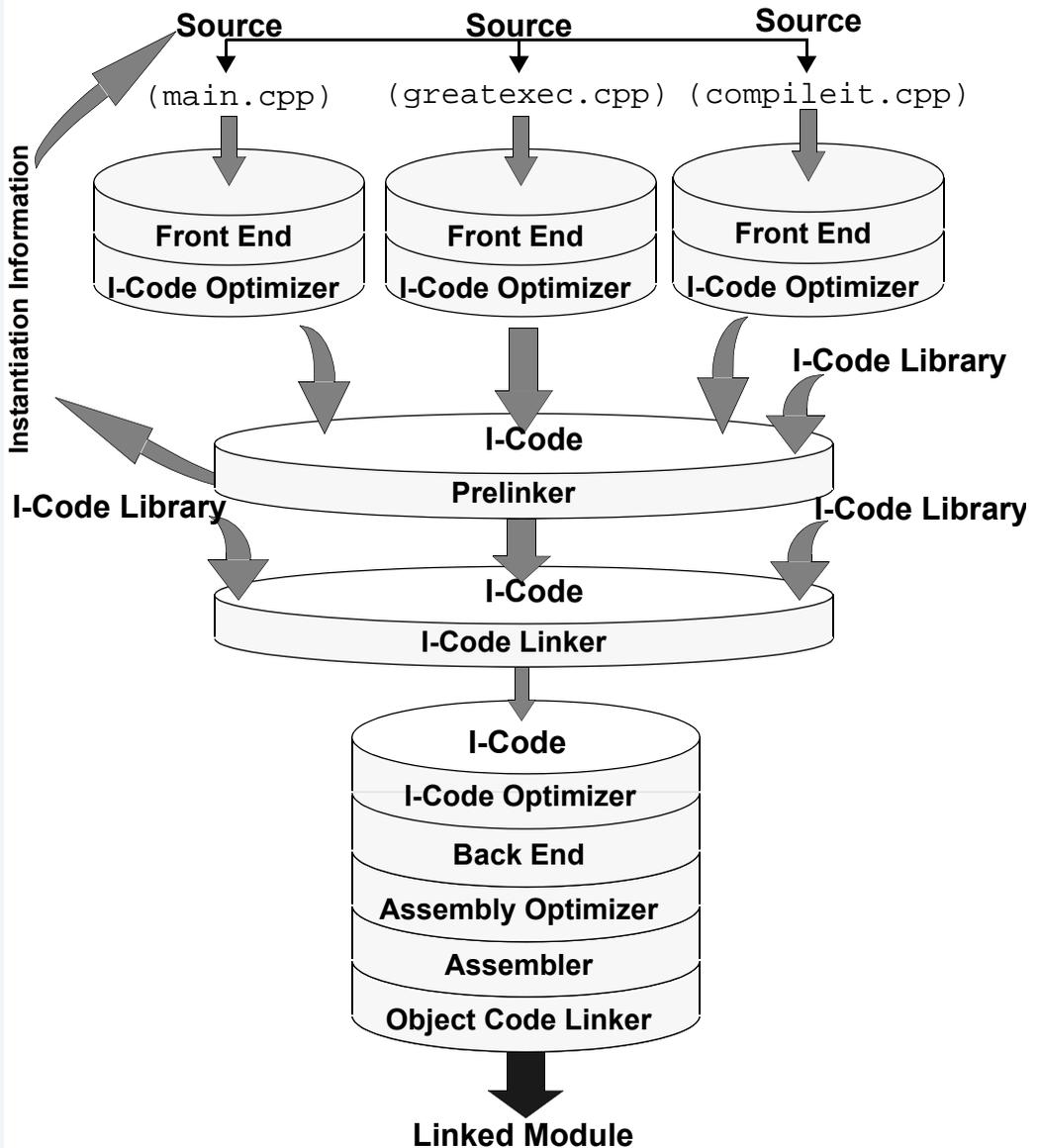
Figure 1-3 I-Code Link Method for Compiling Multiple C Source File Applications



Each C source file is compiled to its I-code file, linked with other I-code files and libraries, and processed as a single file to produce an OS-9 module.

I-code link processing of C++ source files is illustrated in the following Figure.

Figure 1-4 I-Code Link Method for Compiling Multiple C++ Source File Applications



Each C++ source file is compiled to its I-code file. Templates are automatically instantiated when:

- linking object and I-code files.
- linking with other I-code files and libraries.
- processing as a single file produces an OS-9 module.

Object Code Linker

With the object code link method, each file is compiled to its ROF, and all ROFs are then linked to produce an OS-9 module.



For More Information

ROFs are defined in detail in [Chapter 6](#).

Compiling source files with the object code link method produces quality code faster than with the I-code link method. Each source file is fully optimized, but source files are not optimized with respect to one another. This greatly speeds the time required to compile programs because only files that have been changed since the last compilation require optimization. This is the model recommended for use in application development.

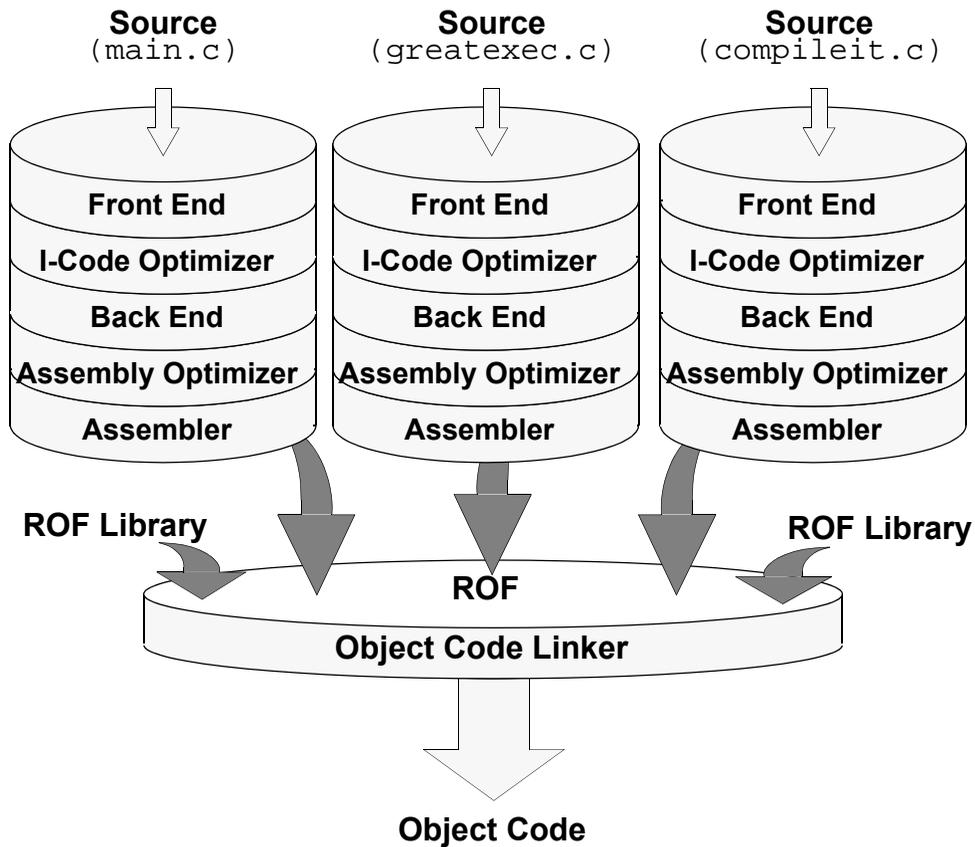
To use the object code link method, enter a command line similar to the following:

```
xcc -td=c:\temp main.c greatexec.c compileit.c -f=prog -x=il  
-td=c:\temp and -f=prog are options passed to the compiler.  
main.c, greatexec.c, and compileit.c are source code files to  
be linked.
```

The `-x=il` option is the most important feature of this command line. It directs the compiler to skip the I-code Linker phase and link only files at the object level.

Compiling C source files with the object code link method is illustrated in the following figure.

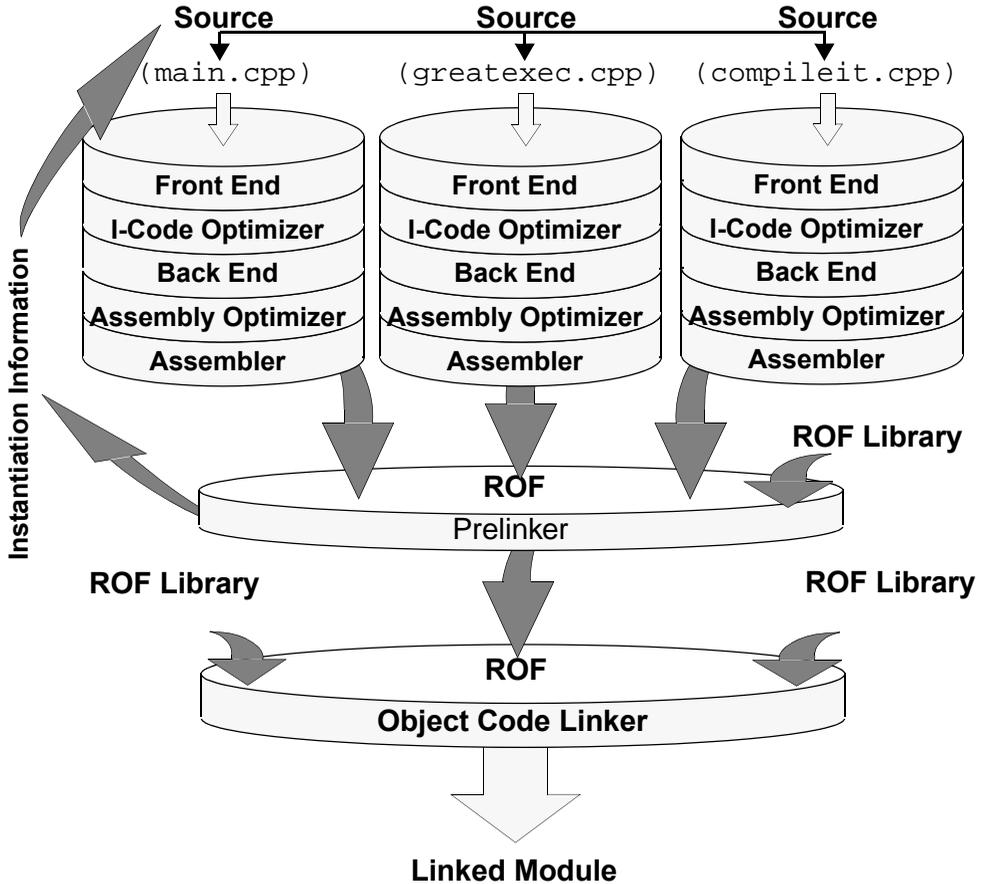
Figure 1-5 Object Code Link Method for Compiling Multiple C Source File Applications



Each C source file is compiled to an ROF and linked with other ROFs and libraries to produce an OS-9 module.

Compiling C++ source files with the object code link method is illustrated below.

Figure 1-6 Object Code Link Method for Compiling Multiple C++ Source File Applications



Each C++ source file is compiled to an ROF, templates are automatically instantiated, and the ROF and templates are linked with other ROFs and libraries to produce an OS-9 module.

Source Code Compatibility

In addition to the executive option modes, five source code compatibility modes exist as identified following.

- **K & R** source mode exists for compatibility with the Microware K&R C compiler. Refer to **Chapter 2** for information concerning incompatibilities between C/C++ and the Microware K&R C compiler.
- **ANSI** source mode compiles only programs conforming strictly to the ANSI/ISO standard as described in the standard. ANSI source mode tracks the standard exactly to provide portability of standard-conforming applications.
- **Extended ANSI** source mode enables running of programs that use extensions to the ANSI/ISO standard. This mode generally follows the ANSI/ISO standard except that features may be added to enhance the OS-9/compiler environment in the future.
- **c front 2.1** duplicates a number of features and bugs of `cfront`. Complete compatibility is not guaranteed or intended. `c front 2.1` mode exists to enable compilation of existing code which implements `cfront` features.
- **c front 3.0** duplicates a number of features and bugs of `cfront`. Complete compatibility is not guaranteed or intended. `c front 3.0` mode exists to enable compilation of existing code which implements `cfront` features.

Libraries that are linked automatically with the source code is dependent upon the executive source mode used during compilation. Libraries specified as command line options are also linked.

The following table identifies the defaults for each of the `xcc` executive option modes.

Table 1-3 xcc Executive Option Mode Compatibility and Defaults

xcc Executive Option Mode	Source Mode	Default Libraries	Additional C++ Default Libraries
<code>compat</code>	K & R	<code>sys_clib.1</code> <code>clib.1</code> <code>os_lib.1</code> <code>math.1 (OS-9 for 68K only)</code> <code>sys.1 (OS-9 for 68K only)</code>	NA
<code>c89</code>	ANSI	<code>clib.1</code> <code>os_lib.1</code> <code>sys.1 (OS-9 for 68K only)</code>	<code>cplib.1/</code> <code>cplibnx.1</code>
<code>ucc</code>	Extended ANSI	<code>clib.1</code> <code>os_lib.1</code> <code>sys.1 (OS-9 for 68K only)</code>	<code>cplib.1/</code> <code>cplibnx.1</code>

Chapter 2: Compiling

This chapter includes information on changing compiler defaults and controlling aspects of compilation. It includes the following sections:

- **main() Function Parameters**
- **Code Size**
- **Code Speed**
- **Time and Space Control**
- **Floating Point Math**
- **Assembly Language in C Source Files**
- **Stack Checking**
- **Multi-Threading**
- **Keywords**
- **C++ Features and Restrictions**
- **Object Size and Alignment**
- **Compatibility with the Microware K&R C Compiler**
- **Running Compiler Makefiles**
- **Differences Between Compatibility Source Mode and the Microware K&R C Compiler**



main() Function Parameters

The compiler provides three `main()` function parameters:

- `argc`
- `argv`
- `envp`

`argc` and `argv` are standard C/C++ language parameters. `envp` is a pointer to the base of a NULL terminated list of environment variables and their values. This list is also pointed to by `_environ` (for ANSI compilations only) and `environ` (for non-ANSI compilations only).

`main()` might be declared as:

```
int main(int argc, char *argv[], char *envp[])
```

Each string pointed to by entries of the environment list is formatted as:

```
<name>=<value>
```

For example:

```
"PATH=/h0/CMDS"
```

To enlarge the list, allocate a new list and copy the pointers from the original list to the new list.

Code Size

The compiler generates code that works for any size application by default. To produce the smallest code, override the compiler defaults.

Defaults

Because the compiler code generation ensures compilation and execution of any size program, all external references (for example, calls to functions outside the current file) are generated in the long form.



For More Information

Refer to the *Ultra C/C++ Processor Guide* for long and short form values specific to a processor.

Methods for Reducing Compiled Code Size

Methods for reducing compiled code size include overriding compiler reference size defaults, I-code linking, and using provided, small library functions.



For More Information

For information on overriding compiler reference size defaults, refer to the appropriate processor chapter in the *Ultra C/C++ Processor Guide*.

I-Code Linking

To create the smallest compiled production code:

- compile all code to I-code
- link I-code with the I-code libraries

I-code linking greatly decreases the number of external references that are accessed using the long form displacements. By using the `-j` executive option to I-code link with the libraries, the code from the libraries may be much smaller because:

- Object code libraries are generated using long-form displacements for external library references.
- I-code linking enables the maximum number of references to be internal and does not require long-form addressing of internal references until the displacement is too large to be reached by short-form addressing.

Small Library Functions

Functions in the standard C library conform to the ANSI/ISO standard, containing information such as locale and multi-byte character set. This conformity greatly increases the size of the functions. As a result, when certain functions are used, the size of code greatly increases. To minimize code size, smaller versions of some library functions are available in the libraries `sclib.l` and `sclib.il`. To accomplish the reduction, slower algorithms were used or functionality was removed. The `sclib.l` and `sclib.il` libraries comprise the following functions:

<code>exit()</code>	<code>fclose()</code>	<code>fprintf()</code>	<code>fread()</code>
<code>fscanf()</code>	<code>fseek()</code>	<code>fwrite()</code>	<code>printf()</code>
<code>scanf()</code>	<code>sprintf()</code>	<code>sscanf()</code>	<code>vfprintf()</code>
<code>vprintf()</code>	<code>vsprintf()</code>		

Table 2-1 identifies differences between each small library function and the standard C library function for which the function substitutes.

Table 2-1 Small Library Function Differences

Small Library Function	Difference
<code>exit()</code>	Does not work with <code>atexit()</code>
<code>fclose()</code>	Does not delete temporary files created by <code>tmpfile()</code>
<code>fread()</code>	Smaller and slower function
<code>fseek()</code>	Smaller and slower function
<code>fwrite()</code>	Smaller and slower function

The following small library functions do not support locale, or multi-byte character set support, or floating point format types (disallowing use of `%e`, `%f`, and `%g`).

<code>fprintf()</code>	<code>fscanf()</code>	<code>printf()</code>	<code>scanf()</code>
<code>sprintf()</code>	<code>sscanf()</code>	<code>vfprintf()</code>	<code>vprintf()</code>
<code>vsprintf()</code>			



Note

To use the functions in the small libraries, use a command line similar to that displayed below:

```
xcc myfile.c -l=sclib.l
```

Code Speed

The compiler generates fully-optimized code by default. Adjusting optimization options may alter the speed of executable code.

Optimization Options

The compiler has eight optimization levels; scaled from 0 to 7, level 0 disables optimization and level 7 enables full optimization. Options controlling the level of optimization performed by executive option mode are:

```
ucc or compat = -o
c89           = -O
```

The default optimization level for the compiler is level 7 in `ucc` and `c89` option modes (`-o=7` and `-O=7`, respectively) and level 1 in `compat` option mode.

Normally, changes to the default optimization levels are not required. If an optimization override is necessary (due to hardware register value changes not visible to the compiler or to avoid a compiler bug), lower the optimization level to disable specific optimizations. The `-o` levels (for `ucc` option mode) and `-O` levels (for `c89` option mode) are provided for optimization control without switching specific optimizations on or off.

Maximum control of optimizations occurs by passing options directly to the I-code optimizer and other components by using `-io<option>`. For example, to direct the optimizer to automatically inline single call functions and discard the function after inlining, enter the following command:

```
xcc -iom test.c
```

The I-code optimizer `-m` option may be used when a function is called only once internal to the file.



For More Information

Chapter 5: Compiler Phase Options, lists the options passable to the I-code optimizer from the command line. **Chapter 11**, describes optimizations that the compiler performs.



Note

Optimization controls such as `-o` and `-io<option>` that degrade the default (full optimization) should not be required. The ANSI/ISO C/C++ volatile keyword functionality (described later in this chapter) eliminates the possibility of erroneous code as a result of optimization.

Time and Space Control

The compiler provides options to control requirements for the amount of time required to execute and the size of the final object code file.



Note

The `ucc` option mode options `-s` and `-t` are used in examples in this section.



For More Information

Chapter 11, contains additional information about partial and full loop unrolling, automatic function inlining, and common tail merging.

Time and space options are identified in [Table 2-2](#).

Table 2-2 `ucc` and `c89` Option Mode Time and Space Options

<code>ucc</code>	<code>c89</code>	Specifies the Importance Of
<code>-s [=] <num></code>	<code>-n <num></code>	A small code size for the file
<code>-t [=] <num></code>	<code>-m <num></code>	A fast execution speed of the file

Time and space factors may be thought of as multipliers. Refer to the code below as an example:

```
ucc -t=10 test.c
```

The above command directs the compiler to place ten times more importance on time than on space. Using this command may cause the compiler to make the program larger to gain extra speed. The maximum value for time or space is 10.

Time and space controls the type of code generation and optimizations performed. The time option controls optimizations such as:

- Full or partial loop unrolling
- Automatic function inlining
- Code selection in the back end



Note

Raising the time factor excessively can lead to large code size and slow compilation and can, past a point, be counter-productive. For example, larger code may mean more instruction cache misses.

The space option controls:

- Code selection
- Common tail merging

Maximizing Speed

Use the following command to maximize speed:

```
xcc -s=0 test.c
```

This indicates that space is of zero importance, enabling the compiler to inline every function and unroll every loop possible.

Minimizing Space

To minimize space, use the following command:

```
xcc -t=0 test.c
```

This indicates that time is of zero importance, enabling the compiler to make the program as slow as possible if doing so yields smaller code.

Using Time/Space Ratios

Ratios may be used to obtain non-integer time/space balances:

```
xcc -t=3 -s=2 test.c
```

This means that time is of 1.5 times greater importance than space.

Floating Point Math

The compiler may support one of two different floating point models, depending upon target architecture. The two models are referred to as

- Hardware Emulation
- Software Emulation

Hardware Emulation is the most common model used by the compiler.

Hardware Emulation

With Hardware Emulation, the compiler generates only inline floating point instructions, and the C library uses inline floating point instructions for its routines that require that functionality. A floating point emulation module is used on systems without sufficient floating point hardware. This model catches unimplemented instruction (or the equivalent) traps and emulates floating point hardware. By always generating inline floating point code, the compiler creates a consistent interface for applications running on machines with or without floating point hardware, minimizes code size, and provides the greatest performance on machines with floating point hardware.

Software Emulation

With Software Emulation the compiler uses statistically linked floating point emulation libraries to perform floating point operations. This model results in much larger code generation and is only provided if a consistent definition of floating point hardware and instruction set is available.

Assembly Language in C Source Files

In some cases, only assembly language is suitable for a program.

1. System-level code may have to use instructions that the compiler never emits to implement mutual exclusion, perform I/O, or modify the processor state in some way beyond the scope of the C and C++ language
2. Optimization may require the use of assembly language to take advantage of a peculiarity of the target processor.

For this reason, the compiler allows the inclusion of assembly language in C source code with statements that look like calls to a function returning void named `_asm()`. These statements can appear either within a function or outside a function. `_asm()` statements outside functions, here called external, are useful for writing entire functions in assembly language, defining constants in the code area, or defining global storage areas. `_asm()` statements with C functions are called embedded in this manual. The permissible parameters for embedded `_asm()` statements differ from those allowed in external `_asm()` statements, as does one detail of syntax.



For More Information

Refer to the [External `_asm\(\)` Pseudo Function](#) subsection for a description of the general form `_asm()` statement.

Refer to the [External `_asm\(\)` Statements](#) and [Embedded `_asm\(\)` Statements](#) subsections in this chapter for detailed definitions of syntax and limitations specific to these statements.

Example

```

/* asm.c */

int foo(int x, int y)
{
    int    result;
#ifdef _MPFPPC
    _asm(" add %2,%0,%1",
        __reg_gen(__obj_assign(result)),
        __reg_gen(x),
        __reg_gen(y));
#elif defined(_MPF68K)
    _asm()
        move.l %0,%1
        add.l %2,%1
    ",
        x,
        __reg_data(__obj_assign(result)),
        Y);
#else
    /* We provide a portable version to drop back to while*/
    /* waiting for, or to use instead of, a target-specific*/
    /* version. For a real life function, this may not always be */
    /* possible, but it is desirable.*/
    result = x + y;
#endif
    return result;
}

/* cplusplus.cpp */
#include <iostream>
extern "C"
{
    int foo(int, int);
}

main()
{
    count << foo(1, 2) << endl;
}

```

External `_asm()` Pseudo Function

The general syntax of an external `_asm()` pseudo function is:

```
_asm( <string constant>,
      { <expression>{,<expression>} } )
```

The string constant of the external `_asm()` statement is a character sequence appropriate for assembly by the platform-specific assembler provided by Microware. Double quotes or backslashes within the string, must be preceded with a backslash.

`_asm()` statement string constants may contain elements including carriage returns, labels (specifically placed), format escapes, expression types, etc. Descriptions and examples are provided in this subsection for each of the element in the `_asm()` syntax.



Note

Source formats used for examples in this section reflect the Microware format. Variance from the shown formats is acceptable.

Table 2-3 Sample `_asm()` Statements

<code>_asm()</code> Statement	Description
<code>_asm(" vsect");</code>	Assembler directive
<code>_asm("var: ds.1 1");</code>	Global storage declaration
<code>_asm(" ends");</code>	Global storage declaration
<code>_asm("const: dc.1 \$30");</code>	Code area constant definition
<code>_asm("* external function");</code>	Assembly language comment

Table 2-3 Sample `_asm()` Statements (continued)

<code>_asm()</code> Statement	Description
<code>_asm("Init:");</code>	Code area function label
<code>_asm(" move.l const(pc),d0");</code>	Assembler instruction
<code>_asm(" move.l d0,var(a6)");</code>	Assembler instruction
<code>_asm(" rts");</code>	Assembler instruction

String constants may contain carriage returns, providing the programmer with a variety of formatting options. The `_asm()` statements shown in **Table 2-3** may be combined using carriage returns into the following single `_asm()` statement:

```
_asm("
    vsect
var:   ds.l   1
      ends
const: dc.l   $30

* external function
Init:
    move.l const(pc),d0
    move.l d0,var(a6)
    rts
");
```

The string constant must begin with either a label (for example, `var`, `const`, `Init`, etc.) or a space. Assembler directives and mnemonics must be preceded by a space. Labels, if used, must begin in the first character position of the string or after a new line.

```
_asm("label1:           *label is first character of string
    move.l d0,d1
label2                 *label starts in first column of line
    move.l a0,a1
");
```

To enable expressions at the level of C source code to be referenced inside the `_asm()` statement, the string constant may contain format escapes in the form `%<n>`, where `<n>` is a decimal integer. In the assembly generated by the back end of the compiler, the escape is replaced by the result of the `<n>`th expression given in the expression list. Expression numbering begins with 0, so the first expression given in the expression list is referenced using the format escape `%0`.

Example

```
asm( "
    move.l %0,%3      * expr1 -> d0
    add.l  %1,%3      * d0 += expr2
    sub.l  %2,%3      * d0 -= expr3
",
    expr1,           /* %0 */
    expr2,           /* %1 */
    expr3,           /* %2 */
    __reg_d0()      /* %3 */
);
```

The text of the string constant is not read or evaluated by the compiler format escapes. Checking and reporting of errors in the generated assembly language occurs when the back end output is read by the assembler.

Expression types allowed in the expression list of the `_asm()` statement differ between external and embedded statements. The number of format escapes that may be used in one `_asm()` statement is limited only by available memory.

External `_asm()` Statements

Any numeric constant expression may be used in the expression list for an external `_asm()` statement. All other types of expressions are disallowed. When constant numeric expressions are used in an external `_asm()` statement, the decimal integer sequence representing the constant is substituted for the corresponding format escape in the string constant. The following example shows several cases of using numeric constant expressions in an external `_asm()`.

Example

```
#include <stddef.h>
#define VALUE    0x100

typedef struct thing {
    int a;
    int b;
    char c;
} thing;
_asm( "
    vsect
mystruct:      ds.b    %0        * thing mystruct;
myvar:         dc.l    %1        * int myvar = VALUE;
    ends

myfun:
    move.l  #%2,mystruct+%4(a6)    * mystruct.b = 3*4;
    move.b  #%3,mystruct+%5(a6)    * mystruct.c = 'c';
    move.l  #%0,d0                *return(sizeof(thing));
    rts                               * }
    ",
    sizeof(thing),                /* %0 - structure size */
    VALUE,                        /* %1 */
    3 * 4,                        /* %2 */
    'c',                          /* %3 - numeric byte constant */
    offsetof(thing, b),           /* %4 - offset of 2nd field of
                                   /*structure */
    offsetof(thing, c)            /* %5 - offset of 3rd field of
                                   /* structure */
);
```

Code generated by the back end for the previous example follows.

```
vsect
mystruct:      ds.b      10      * thing
mystruct;
myvar:         dc.l      256     * int myvar = VALUE;
ends

myfun:         * myfun()
{
  move.l #12,mystruct+4(a6) * mystruct.b = 3*4;
  move.b #99,mystruct+8(a6) * mystruct.c = 'c';
  move.l #10,d0 * return(sizeof(thing));
  rts *
}
```



Note

Numeric constants are substituted into the string constant as decimal integer sequences in the code generated by the back end. Thus, hexadecimal constants used in the expression list are converted to decimal integers in the assembly.

Embedded `_asm()` Statements

Code in external assembly functions, either in assembly source files or declared in C source with an external `_asm()` statement, is not seen or optimized by the I-code optimizer, nor can such code be inlined. It is, therefore, desirable to place as much of this code as possible within C functions.

The embedded `_asm()` statement enables the programmer to insert assembly code inside a function written in C. It also enables rewriting of external assembly routines as C functions containing only assembly instructions, enabling the I-code optimizer to view the routines as I-code routines and perhaps inline them for increased efficiency.

The syntax of the embedded `_asm()` statement is shown below:

```
_asm( {<size>}, <string constant>,  
      {<expression>{,<expression>}} )
```

The optional size parameter communicates to the I-code optimizer the effect that the `_asm()` statement has on the size of the C function in which it is embedded. The I-code optimizer uses the information to determine which functions should be inlined given the consideration placed on time and space for the compilation.

The size parameter must be a numeric constant and should typically approximate the number of code bytes generated for the embedded assembly instructions. This enables proper representation of the increase in size that inclusion of the `_asm()` causes the C function. One can manipulate the size parameter to influence the optimizer's behavior; a smaller-seeming piece of code is more likely to induce inlining or a greater degree of loop unweaving.

It is impossible for you to know where the compiler chooses to store objects defined in the C code. When using the inline `_asm()` statements, all C objects referenced in the embedded assembly are accessed by format escapes and the corresponding expression list.



For More Information

Additional information on the use of the size parameter is provided in the [Using the Size Parameter](#) subsection later in this chapter.

There are four general types of expressions allowed in embedded `_asm()` statements:

- **C Expressions**
- **Object Usage Pseudo Functions**
- **Register Designator Pseudo Functions**
- **Label Pseudo Functions**

C Expressions

Any valid C expression may be used as an argument for an embedded `_asm()` statement. When the back end finds a format escape in the string constant for an embedded `_asm()`, it replaces the format escape with the character sequence appropriate for addressing the expression stored by the compiler at that point in the code. For example, if a global variable `zz` is referenced as an expression in an embedded `_asm()`, the back end may substitute the effective address `zz(a6)` for the corresponding format escape. If `zz` is frequently used in the C source, the compiler may allocate a register to store the frequently used value. In this case, the back end substitutes the effective address `d4`.

When numeric constant expressions are used in embedded `_asm()` statements, their effective address is an immediate. Substitutions involving constant expressions automatically include the `#` immediate operand designator. This differs from the treatment of constant expressions in external `_asm()` statements.

The following example illustrates the occurrence of effective address substitution within an embedded `_asm()` statement.

Example

```

#define FRED      0x100

int a = 20;

fun()
{
    volatile int b = 30;
    register int c = 40;
    int accum;

    /* _asm to compute 'accum = FRED + a + b + c + (a * 2) ' */
    _asm( "
        move.l    %0,%5
        add.l     %1,%5
        add.l     %2,%5
        add.l     %3,%5
        add.l     %4,%5
        ",
        FRED,      /* %0 - numeric constant */
        a,         /* %1 - global variable */
        b,         /* %2 - local stack variable */
        c,         /* %3 - local register variable */
        a * 2,     /* %4 - arithmetic expression */
        __reg_d0(__obj_assign(accum))
    );
}

```



For More Information

Refer to the [Object Usage Pseudo Functions](#) and [Register Designator Pseudo Functions](#) sections in this chapter for descriptions of `_obj_assign` and `__reg_d0`, respectively.

Sample assembly generated for this `_asm()` statement follows.

```
move.l  a(a6),d7
add.l   d7,d7* (a * 2)

move.l  #0x100,d0* immediate value
add.l   a(a6),d0* global variable
add.l   -4(a5),d0* stack variable
add.l   d1,d0* register variable
add.l   d7,d0* arithmetic expression
```

Object Usage Pseudo Functions

The portions of the compiler up to and including the back end do not understand assembly language, so that they count on you to describe side effects to any C objects referenced in embedded `_asm()` statements. If you do not, the compiler generates code as if the side effects do not occur, which gives incorrect results.



WARNING

Failure to properly communicate to the compiler the side effects to C objects occurring in the `_asm()` statement leads to erroneous results.

Object usage pseudo functions enable communication of side effects and their suggested handling to the compiler. The pseudo functions require a single C identifier as a parameter. There are three such object usage pseudo functions:

```
_ _obj_modify()
_ _obj_assign()
_ _obj_copy()
```



Note

Some pseudo functions in this chapter include a double underscore. To better distinguish between a single and double underscore, an underscore followed by a space and another underscore (`_ _`) appears in this manual to represent a double underscore. The space between underscores is **not** part of the syntax.

A special object usage pseudo function informs the compiler when a numeric constant expression is used in the embedded `_asm()` as something other than an immediate value. This pseudo function requires a single numeric constant parameter.

```
__obj_constant( )
```

**Note**

The examples are intentionally trivial for clarity. Likewise, the code generation depicted appears without benefit of optimization. In a true compile, the code generation is optimal but less illustrative.

An object pseudo function description and use example follows.

Assumptions:

- `a` and `b` are global variables.
- The back end, in each case, moves the object `b` into register `Rx` for use by the embedded `_asm()` statement.

`__obj_modify(x)`

`__obj_modify(x)` indicates that `x` contains a meaningful input going into the `_asm()` which is modified by side effects of the embedded assembly.

Example

```

/* _asm() simulating the C statement 'b = b + a */
_asm( "
    add.l    %0,%1                * b + a --> b
    ",
    a,                                /* %0 - read only access of a */
    __obj_modify(b)                /* %1 - side effect modifies b */
);
}

```

Code generated

```

move.l b(a6),Rx    /* original value of b is read */
add.l  a(a6),Rx    /* side effect modifies b */
move.l Rx,b(a6)   /* modified value of b is written */

```

__obj_assign(x)

`__obj_assign(x)` indicates that `x` is not initialized or contains a value no longer needed going into the `_asm()` and that `x` is overwritten with a meaningful output value.

Example

```

/* _asm() simulating the C statement 'b = a' */
_asm( "
    move.l  %0,%1                * a --> b
    ",
    a,                                /* %0 - read only reference to a */
    __obj_assign(b)                /* %1 - assignment side effect */
);

```

Code generated

```

    /* original value of b ignored */
move.l a(a6),Rx* new value assigned to b
move.l Rx,b(a6)* new value of b is written

```

__obj_copy(x)

`__obj_copy(x)` indicates that `x` contains a meaningful input value for the `_asm()` which should be copied to avoid corruption by any side effects inherent in the embedded assembly.

Example

```
/* _asm() simulating the C statement 'a = b + 1' */
_asm("
    addq.l  #1,%0                * b + 1
    move.l  %0,%1                * b + 1 --> a
    ",
    __reg_data(__obj_copy(b)) /* %0 - copy b before using */
    __reg_data(__obj_assign(a) /* %1 - a is assigned a new */
                /* value */
);
```

Code generated

```
move.l  b(a6),Rx* original value of b is copied
addq.l  #1,Rx* side effect corrupts copy
move.l  Rx,a(a6)* corrupt value not written to b
```



Note

`__obj_copy` requires the programmer to specify which register type is to be used. See the *Ultra C/C++ Processor Guide* for the given target for further details on its use.

__obj_constant(x)

`__obj_constant(x)` indicates that constant `x` is used in such a way that raw substitution of the numeric value is desired.

Example



WARNING

Defining `p` to be a register pointer does not guarantee that it resides in an address register. For this example, assume that it does. The next section in this chapter, **Register Designator Pseudo Functions**, discusses ways to ensure this use.

```
char array[20];
fun()
{
    register char *p = array;

    * __asm() simulating the C statement 'p[5] = 10'
    __asm("
        move.l  %0,%1(%2)
        ",
        10,                /* %0 - used as an immediate */
        __obj_constant(5), /* %1 - to be used without '#' */
        p                 /* %2 - read only access of p */
    );
}
```

Code generated

```
move.l  #10,5(Ax)      * where Ax was selected for p
```

Register Designator Pseudo Functions

Frequently used objects residing in registers enables faster execution. In addition, many instructions require their operands to be in either specific registers or registers belonging to a certain class. The register designator pseudo functions are used to convey these requirements to the code generator. These pseudo functions are platform dependent.



For More Information

Refer to the *Ultra C/C++ Processor Guide*, `_asm() Register Pseudo Functions`, section for register designator pseudo functions supported for your processor.

Register designator pseudo functions optionally accept one parameter. When a parameter is specified, the code generation assures that the value of the specified parameter is moved into an appropriate register prior to entering the embedded assembly code.

The parameter of a register designation pseudo function may be any valid C expression or an object usage pseudo function acting on an appropriate identifier/constant object. If a parameter is not specified for a register designation pseudo function, an appropriate register is allocated for as an unnamed temporary variable.

The following examples illustrate correct and incorrect ways of writing an `_asm()` statement to compute `a = (a << 1)` where `a` is a global variable.

Example 1 (incorrect)

```
_asm( "
    lsl.l    #1,%0
    ",
    a                /* C expression */
);
```

Code generated

```
lsl.l  #1,a(a6)      * illegal addressing mode for lsl
                    instruction!
```

Example 2 (incorrect)

```
_asm( "
    lsl.l  #1,%0
    ",
    __reg_data(a)      /* C expression accessed from data */
                    /* register */
);
```

Code generated

```
move.l  a(a6),Dx    * a is read into some data register
lsl.l  #1,Dx        * addressing mode is correct but
                    * modified value NOT written back
```

Example 3 (correct)

```
_asm( "
    lsl.l  #1,%0
    ",
    __reg_data(__obj_modify(a)) /* C expression being */
                    /* modified by */
);
                    /* side effect while in data reg */
```

Code generated

```
move.l  a(a6),Dx    * a is read into some data register
lsl.l  #1,Dx        * addressing mode is correct
move.l  Dx,a(a6)    * modified value is written back to a
```

The following code provides examples of C expression and object and register selection pseudo function use in an embedded `_asm()` statement.

Given global definitions:

```
int     a, b, c;
int     array[10];
#define     ADJUST         2
#define     INDEX         4
```

```

/* asm simulating 'a += b << (++c + ADJUST - */
   /* array[INDEX])' */

__asm( "
    move.l   %4,%6           * ADJUST --> temp
    sub.l   %5(%3),%6       * temp - array[INDEX] -->temp
    addq.l  #1,%2           * c + 1
    add.l   %2,%6           * temp + (c + 1) --> temp
    lsl.l   %6,%1           * b << temp
    add.l   %1,%0           * a + (b << temp) --> a
    ",
    __obj_modify(a),        /* %0 - value of a is modified */
    __reg_data(__obj_copy(b)), /* %1 - The value of b is needed */
                               /* in a data register. That value */
                               /* is corrupted by our code so a */
                               /* copy of b is used. */
    __reg_d0(__obj_modify(c)), /* %2 - The value of c is */
                               /* referenced from d0 but the */
                               /* initial value is used, and side */
                               /* effects are written back */
    __reg_addr(array),     /* %3 - The address of our array */
                               /* is needed in an address register */
    ADJUST,                /* %4 - Numeric assembly constant */
    __obj_constant(INDEX /* sizeof(int)), */
                               /* %5 - Constant used as an offset */
                               /* in the assembly code */
    __reg_data()           /* %6 - Data register allocated as */
                               /* temporary accumulator */
);

```

The code generated by the prior `_asm()` resembles the following, where `dx` and `dz` represent arbitrarily chosen data registers (`d1 - d7`) and `ay` represents an arbitrarily chosen address register (`a0 - a4`).

```

movem.l dx/d0/ay/dz,-(sp)    * save required registers
    ...
move.l  b(a6),d              * copy b into data register
move.l  c(a6),d0            * move c into d0
lea.l   arry(a6),ay         * load address register

move.l  #2,dz               * ADJUST --> temp
sub.l   16(ay),dz           * temp - arry[INDEX] --> temp
addq.l  #1,d0               * (c + 1)
add.l   d0,dz               * temp + (c + 1)--> temp
lsl.l   dz,dx               * b << temp
add.l   dx,a(a6)           * a + (b << temp)--> temp

move.l  d0,c(a6)           * store modified value of c
    ...
movem.l (sp)+,dx/d0/ay/dz   * restore registers

```

Just as the compiler must be notified of side effects to C objects referenced in embedded `_asm()` statements, it must also be told about register usage.



WARNING

Failure to properly communicate register usage in embedded `_asm()` statements to the compiler leads to erroneous results.

The following is an **incorrect** version of the example used to demonstrate the usage of format escapes:

```
/* d0 used here without setting the compiler know about it. */
/* The compiler cannot think that it is safe to use d0 to */
/* keep some important value across this code.*/

_asm( "
    move.l  %0,d0      * d0 = expr1
    add.l   %1,d0      * d0 += expr2
    sub.l   %2,d0      * d0 -= expr3
",
    expr1,
    expr2,
    expr3
);
```

The recommended access is similar for arrays, as shown in the following example.

Example

```
int a,b,c;
int array[10];
#define ADJUST 2
#define INDEX 4
_asm( "
    move.l  %4,%5
    sub.l   %3,%5
    addq.l  %1,%2
    add.l   %2,%5
    lsl.l   %5,%1
    add.l   %1,%0
    ", _ _obj_modify(a),
    _ _reg_data (_ _obj_copy(b)),
    _ _reg_d0 (_ _obj_modify(c)),
    _ _reg_data(array[INDEX]),
    ADJUST,
    _ _reg_data()
);
```

Assembly language code generated

```

movem.l dz/dy/dx/dw/do,-(sp)
    ...
move.l =arry+16(a6),dz
move.l =c(a6),d0
move.l =b(a6),dy
moveq.l #0x2,dx

move.l dx,dw
sub.l dz,dw
addq.l #1,d0
add.l d0,dw
lsl.l dw,dy
add.l dy,=a(a6)

move.l d0,=c(a6)
    ...
movem.l (sp)+,dz/dy/dx/dw/d0

```

Label Pseudo Functions

Hard-coding label names in the string constant of embedded `_asm()` statements is not recommended, as C functions may be inlined in multiple places in a calling routine.

Flow control within embedded `_asm()` statements is supported through the `__label()` pseudo function. This pseudo function does not accept parameters. When `__label()` appears in the expression list for an embedded `_asm()` call, the code generator allocates a unique assembly label and substitutes that label for the corresponding escape in the string constant. The escape defining the position of the label within the string constant must reside as the first character position of the string or at the first character immediately following a carriage return.



WARNING

Symbolic labels defined inside embedded `_asm()` statements result in duplicate symbol errors from the linker.

Labels may only be accessed from within the `_asm()` statement in which they are positioned.

Attempts to branch from one embedded `_asm()` statement to another can lead to erroneous results, because the I-code optimizer does not notice the branch.

Given global definitions

```
int a, b;
```

Example 1

```
/* asm simulating the simple loop */
do {
    a = a - 2
} while (a);

_asm( "
%1
    subq.l  #2,%0      * a = a - 2
    bne.b  %1         * while (a)
    ",
    __obj_modify(a), /* %0 - a      */
    __label()       /* %1 - loop label */
);
```

Code generated

```
__$L2
    subq.l  #2,a(a6)   * a = a - 2
    bne.b  _$L2       * while (a)
```

Example 2

```

/* this _asm() statement determines the bit number of the */
/* least significant set bit in a, and stores it in b. */
/* If a = 0, the _asm() stores -1 in b */

_asm( "
    move.l  #-1,%1          * initialize b to -1
    tst.l   %0              * test value of a
%2  addq.l  #1,%1          * increment bit count at
                               beginning of loop
    lsr.l   #1,%0          * shift out lowest bit
    bcc.b   %2              * repeat loop if bit was not set
%3
    ",
    __reg_data(__obj_copy(a)), /* %0 - lsl requires data reg */
    __reg_data(__obj_assign(b)), /* %1 - use data reg for speed */
    __label(), /* %2 - loop label */
    __label() /* %3 - exit label */
);

```

Code generated

```

    move.l  a(a6),Dx

    move.l  #-1,Dx
    tst.l   Dx
    beq.b   _$L2
_$L3
    addq.l  #1,Dx
    lsr.l   #1,Dx
    bcc.b   _$L3
_$L2
    move.l  Dx,b(a6)

```

Using the Size Parameter

The overhead of parameter passing, calling, and returning from a function can be expensive in terms of time. For very small functions, this overhead can also be seen as increased code size. The I-code optimizer tries to inline functions where they are called if it sees benefit given the time and space considerations specified on the command line.

Since the I-code optimizer does not comprehend the assembly language of the `_asm()` statement, it cannot determine the effect this code has on the C function in which it is embedded. The optional size parameter, as mentioned earlier, enables the programmer to specify the expected code size increase caused by the presence of the embedded assembly code. The size parameter value is added to the calculated byte count of the C function in which it is embedded and that size is used to determine whether the function can be inlined.

Size parameter use example:

```
_asm(2, "nop");
```



Note

If a negative size is given in an `_asm()` statement, the surrounding C function is counted as though it were smaller than it actually is.

Therefore, the `_asm()` statement

```
_asm(-500, "");
```

could be used inside a function that is 500 bytes or less to ensure that the function is inlined.

Regardless of the size of a C function, it may be inlined if it is called only once and the `-iom` option is used during the compile. Larger functions may also be inlined if the `-t` option is used to increase (beyond the default setting) the consideration given to time. For small functions, inlining happens regardless of `-iom`, `-t`, and `-s`. This is due to inlined code always being smaller and faster.

An `_asm()` statement without a specified size is presumed to be large enough to make repeated inlining undesirable under default compiler settings.

The first occurrence of an `_asm()` statement without a size specification within a C function increases the effective size of the function to a point too large to inline given default compiler settings. Subsequent uses of unsized `_asm()` statements in the C function increase the effective size of the function by roughly ten bytes per statement.

Suggestions Related to Using `_asm()` Statements

Suggestions related to using `_asm()` statements discussed in this subsection include:

- [Inlining External Assembly Functions](#)
- [Avoiding Hard-Coded Labels in `_asm\(\)` Statements](#)
- [Referencing Objects Directly](#)
- [Allocating Registers in Embedded Assembly](#)

Inlining External Assembly Functions

The inability of the I-code optimizer to inline external assembly functions makes it preferable to write external assembly functions as C functions using embedded `_asm()` statements.

Example

Consider the following `main()` function (input code) that calls the routines `mask_sr()`, and `restore_sr()`.

```
main()
{
    int old_sr;          /* storage for old status register image */
    old_sr = mask_sr(4); /* mask interrupts to level 4 */
    ...                 /* continue processing */
    restore_sr(old_sr); /* restore original status register */
    ...                 /* continue processing */
    old_sr = mask_sr(2); /* mask interrupts to level 2*/
    ...                 /* continue processing */
    restore_sr(old_sr); /* restore original status register */
    ...                 /* continue processing */
}
```

If the `mask_sr()` and `restore_sr()` routines are defined as external assembly routines by the following external `_asm()` statements, the following code is generated.

```
_asm("
                                * level is passed in d0
                                * old sr is returned in d0

mask_sr:
    move.l  d1,-(sp)             * save register
    move.w  sr,d1               * copy status register
    andi.w  #$f8ff,d1          * clear interrupt mask bits
    lsl.w   #8,d0               * align level select bits
    or.w    d0,d1               * make new status register mask
    move.w  sr,d0               * load current sr for return value
    move.w  d1,sr               * write new status register mask
    move.l  (sp)+,d1           * restore register
    rts

");

_asm("
* sr image passed in d0
restore_sr:
    move.w  d0,sr              * restore old status register image
    rts

");
```

Code generated

```

main:                                * register dx allocated to store old_sr
    ...
    moveq.l #0x4,d0                  * pass level to mask_sr()
    bsr mask_sr                       * call mask_sr()
    move.l d0,dx                      * store return value in old_sr
    ...    move.l dx,d0              * pass old_sr to restore_sr()
    bsr restore_sr                   * call restore_sr()
    ...
    moveq.l #0x2,d0
    bsr mask_sr
    move.l d0,dx
    ...
    move.l dx,d0
    bsr restore_sr
    ...
    rts

mask_sr:
    move.l d1,-(sp)
    move.w sr,d1
    andi.w #$f8ff,d1
    lsl.w #8,d0
    or.w d0,d1
    move.w sr,d0
    move.w d1,sr
    move.l (sp)+,d1
    rts

restore_sr:
    move.w d0,sr
    rts

```

Calls to `mask_sr()` and `restore_sr()` do not require changing to embedded `_asm()` statements to take advantage of I-code optimizations. Rather, rewrite the routines as C functions with embedded `_asm()` statements and allow inlining.

```

mask_sr(level)
int level;
{
    _asm(0,                /* size of statement set to zero to */
        /* encourage inlining */
        "
        move.w  sr,%1
        andi.w  #$f8ff,%1
        lsl.w   #8,%0
        or.w    %0,%1
        move.w  sr,%0
        move.w  %1,sr
        ",
        __reg_data(__obj_modify(level)),
        __reg_data()
    );
    return(level);
}

restore_sr(value)
int value;
{
    _asm(0,                /* size of statement set to zero to */
        /* encourage inlining */
        "
        move.w  %0,sr
        ",
        value
    );
}

```

If this source is compiled, together with the prior `main()` function example (using the option `-iom`, allowing inlining and discarding of inlined functions), the following code is generated:

```

main:                * register dx allocated to store old_sr
...
moveq.l #0x4,dx    * allowed, as old_sr is uninitialized

move.w  sr,d0     * inlined version of mask_sr()
andi.w  #$f8ff,d0
lsl.w   #8,dx
or.w    dx,d0
move.w  sr,dx    * note the automatic assignment into old_sr
move.w  d0,sr

```

```
...
    move.w  dx,sr    * inlined version of restore_sr()
...
    move.w  sr,d0    * inlined version of mask_sr()
    andi.w  #$f8ff,d0
    lsl.w   #8,dx
    or.w    dx,d0
    move.w  sr,dx    * note the automatic assignment into old_sr
    move.w  d0,sr
...
    move.w  dx,sr    * inlined version of restore_sr()
```



Note

Notice the time and space effectiveness resulting from the above scenario.



Note

All functions in the C and OS libraries that are written in assembly language use the above mechanism for their I-code library versions. This mechanism generates the most efficient code for all assembly language functions.

Avoiding Hard-Coded Labels in `_asm()` Statements

Hard-coded labels within embedded `_asm()` statements is not recommended. Following this guideline reduces the possibility of redefined label errors. Always use the `__label()` mnemonic within embedded `_asm()` statements to ensure generation of unique label names.

Referencing the generated code sample on the prior page, the embedded `_asm()` statement contained in the C function `mask_sr()` was inlined in two places in the function `main()`. If a hard-coded label were added in the `_asm()`, the linker would disallow redefined labels in the resulting code and generate the following error message:

```
*** error - redefined label ***
```

Referencing Objects Directly

Perform work at the C level when possible for greatest optimization. In particular, write embedded `_asm()` statements to reference objects as directly as possible. Following is an example of a common mistake. Although the code works, it is less efficient than the second example.

Example (common mistake)

```
int a;

main()
{
    fun(&a);
}

fun(ptr)                /* add 4 to integer pointed to by ptr */
int *ptr;
{
    _asm("
        add.l  #4,(%0) * de-reference of pointer object
        ",
        __reg_addr(ptr)
    );
}
```

Code generated after inlining

```
main:
    ...
    lea.l a(a6),ax      * where ax is chosen for ptr
    add.l #4,(ax)
    ...
```

Since it is the value stored at `ptr` that changes within the `_asm()` statement, use that object as the parameter to the `_asm()` statement rather than `ptr` itself.

Example (efficient)

```
fun(ptr)
int *ptr;
{
    _asm("
        add.l #4,%0          * direct access of object
        ",
        __obj_modify(*ptr) /* dereference at C level*/
    );
}
```

Code generated after inlining

```
main:
    ...
    add.l #4,a(a6)          * the pointer was optimized away
    ...
```

It is also possible to generate incorrect code if you attempt to manipulate too many variables. Instead, allow the compiler to manipulate the pointers. Consider the following examples:

Example (incorrect)

```
struct t
{
    int a;
    int b;
    char *ptr[20];
} a;
main()
{
    fun (&a);
}

fun(x)
struct t *x;
}
    _asm("
        add.l #4,(%0)
        ",
        __obj_modify(x->ptr[10])
    );
}
```

Code generated after inlining

```
main:
    ...
    lea =a(a6),a0
        add.l #4,(0+48(a0))    * This won't work
    ...
```

Example (correct)

```
struct t
{
    int a;
    int b;
    char *ptr[20];
} a;
main()
{
    fun(&a);
}
fun(x)
struct t *x;
{
    _asm("
        add.l #4,%0
        ",
        __obj_modify(*x->ptr[10])
    );
}
```

Code generated after inlining

```
main:
    ...
    lea =a(a6),a0
    move.l 0+48(a0),a1
        add.l #4,(a1)
    ...
```

Allocating Registers in Embedded Assembly



Note

Using the register pseudo-functions to access registers in `_asm()` statements is important even for registers that the compiler dedicates to a specific purpose.

The following example uses the pseudo functions to access registers. If the compiler did not previously provide a means to determine maximum stack usage, writing the function shown is useful in compensating and tracking the stack pointer at various points in the program.

```
void *
get_sp(void)
{
    void*result;
    _asm("
        move.l %0,%1
        ",
        __reg_a7(),
        __reg_gen(__obj_assign(result))
    );
    return result;
}
```

Error Messages

Improperly written `_asm()` statements cause error from any of four different phases of compilation; front end, back end, assembly code optimizer, or assembler.

The front end performs a cursory syntax check of the `_asm()` statement to ensure proper formation of the pseudo function call. Error reporting in this phase is consistent with error reporting for other C statements. The following error may also be generated by the front end.

```
***** error in assembly-language escape *****
```

This error usually indicates incorrect typing of the `<size>` or `<string constant>` parameter of the `_asm()`. For example, each of the following result in this error:

```
_asm(0); /* no <string constant> specified */
_asm(0, 0); /* <string constant> not a string constant type */
_asm(x, ""); /* <size> is not a numeric constant */
```

Errors occurring in the assembly instructions are reported in the assembly phase of compilation. The assembler reports errors as described in [Appendix A](#).

The back end does not provide robust error-checking on semantically incorrect I-code. It does, however, detect improper I-code generated as the result of ill-formed `_asm()` statements and aborts with an error message.

If the string constant of an `_asm()` statement contains a format escape corresponding to an expression not given in the expression list, the back end generates the following error.

```
**** not enough arguments for assembly-language escape ****
```

The following `_asm()` statement results in the above error:

```
_asm(" move.l %0,%1", x); /* one expression given; */
                          /* two referenced */
```

Any other errors encountered by the back end result in the following default error message.

```
**** internal error - no pattern match ****
```

The previous error is most often caused by incorrect spelling of the object usage, register selection, or label generation pseudo functions.

Examples of statements generating this back end error include:

```
_asm("
    tst.l %0
    ",
    __reg_blech() /* incorrect spelling */
);

_asm("
    tst.l %0
    ",
    __obj_modify(2+3) /* illegal parameter type */
```

```
);

__asm( "
    tst.l %0
    ",
    __obj_assign(_ _reg_data(x)) /* illegal nesting */
);

__asm( "
    tst.l %0
    ",
    __obj_copy() /* parameter missing */
);
```

Incorrect spelling of pseudo function names may also produce unresolved references during link if the prefix of the pseudo function is misspelled. Common examples are:

```
_obj_modify(x) /* single '_' */
__regdata() /* missing '_' */
__lable() /* spelling */
```

Stack Checking

At the start of each function that uses stack space, the compiler can generate code ensuring that the program is not out of stack space. This is called stack checking. Stack checking may be disabled if the program is certain not to exhaust stack space or if the stack checking is inappropriate for the module being built. The stack disabling options are displayed below:

```
ucc and c89 modes    =    -r
compat mode         =    -s
```

Non-Program Modules

If a non-program module (a module that does not link with `cstart` as its root `psect`) is created that links with any of the libraries (`cplib.l/cplibnx.l`, `clib.l`, `sys_clib.l`, or `os_lib.l`), provisions for stack checking may be required as some library functions have stack checking enabled. Stack checking is accomplished using five global variables and two functions from `clib.l`. [Table 2-4](#) identifies global variables used by the stack checking functions. [Table 2-5](#) identifies functions used in stack checking and when the functions are called.

Table 2-4 Global Stack Checking Variables

Global Variable	Description
<code>_maxstack</code>	Maximum number of bytes used from the stack
<code>_mtop</code>	Overflow point of stack (lowest address of stack memory)
<code>_stbot</code>	Lowest value the stack pointer has reached

Table 2-4 Global Stack Checking Variables (continued)

Global Variable	Description
<code>_stklimit</code>	Distance from the current stack pointer to the lowest value the stack pointer has reached (<code>_stbot</code>) This global variable is only available on 68K and x86 processors.
<code>_sttop</code>	Value of the stack pointer when the program started

Table 2-5 Stack Checking Functions

Function	Called
<code>_stkoverflow</code>	When a stack overflow is detected (<code>_stbot</code> goes below <code>_mtop</code>)
<code>_stkhandler</code>	When stack needs to be allocated (<code>_stklimit</code> is less than zero or the stack pointer goes below <code>_stbot</code>)

Stack checking of non-program modules requires initialization (with appropriate values) of global variables before the first call is made to a function with stack checking code.



Note

For information on how to disable the effects of existing stack checking, refer to the chapter of *Ultra C/C++ Processor Guide* that is specific to your processor.

Multi-Threading

The compiler's default environment is a non-threaded environment. To compile code for a threaded environment, the `ucc` and `c89` modes provide a `-mt` option, which controls multi-threading options.

Enabling multi-threading affects the following:

- An `_OS9THREAD` preprocessor/assembler macro is defined. This can be used to conditionally compile code for a multi-threaded environment.
- The definitions of various globals change such that they are made thread- safe. This includes the compiler globals declared in `<errno.h>` and `<cglob.h>`.
- Additional options may be passed to the various compiler phases to support multi-threading.
- Modules produced by the compiler are linked against thread-safe libraries instead of the default non-thread-safe libraries.

Libraries

The thread-safe libraries begin with an `mt_` prefix. For example, `mt_clib.l` is the thread-safe version of `clib.l`.

You do not need to explicitly request that a thread-safe version of a library be linked in. All that is necessary is the use of the `-mt` option.

For example, if you specify to link against the `f00.l` library using the `-l=f00.l`, the compiler will first search for `mt_f00.l` (if the `-mt` switch is used) and then search for `f00.l`. The `-mt` switch accepts several options that enable you to more finely control the library searching.



For More Information

The options are described in **Table 3-8** of **Chapter 3**.

The compiler also protects against accidental linking with incompatible libraries and intermediate files. This can happen when a thread-safe library was not found or an intermediate file will not built correctly.

Libraries and intermediate files can fall under one of the following thread-using and thread-safety categories:

- None
- Using, Safe
- Safe

In some cases, libraries are inherently thread-safe. These libraries are marked as "safe", in which case a separate thread-safe version of the library need not exist. All other libraries are marked as "none" or "using, safe".



For More Information

For more information about multi-threading, see *Using OS-9 Threads*.

CSL and Multi-Threaded Applications

The following two CSL (C Shared Library) files are available when configuring OS-9 systems:

- `cs1`
- `mt_cs1`

Both files contain a module named `cs1`. Use the file `cs1` for systems that execute no threaded `cs1`-using applications. Use the file `mt_cs1` for systems that execute both threaded and non-threaded `cs1`-using applications.



Note

The Configuration Wizard contains a **Thread Support** check-box that, when selected, installs the `mt_cs1` file to your system. The default configuration uses the `cs1` file.

Keywords

The following keywords are described in this section:

- **volatile**
- **const**
- **remote**

volatile

The **volatile** keyword specifies that:

- storage for an object may change at any time
- assignments or references to the object explicitly placed in the program must remain

Use `volatile` when the value of an object can change without compiler knowledge. For example, declare all global variables changed in a signal handler as `volatile` as a signal may arrive at anytime and cause the globals to change value. The compiler is oblivious to the change unless the variable is declared as `volatile`.

The `-cg` option for the I-code optimizer may be required when compiling code that is being included in a module with a signal handler or an interrupt service routine. `-cl` suppresses the common sub-expression elimination (refer to [Chapter 5: Compiler Phase Options](#)) containing a local variable with an address stored such that the value of the variable could change without the compiler's knowledge. For example, if the address of a local in a global variable is stored and a signal handler uses that global variable to report a signal's occurrence, problems could occur if the value of the local variable is stored in a temporary register when the signal arrives.



Note

Generally, declaring all variables volatile that are changed by asynchronous events (such as signal handlers and interrupt service routines) ensures that the optimizer does not save the value of any of these variables in registers.

const

The `const` keyword specifies that the storage for an object may not be modified, thereby conserving memory. The compiler allocates storage for global `const` qualified objects in the code space of the module, or in the case of the automatic storage class, on the stack. This allows storage of large static tables in the module, enabling sharing of the tables by all processes using the module, rather than storing the tables in the data space of each process.

Because OS-9 requires position-independent, re-entrant code, `const`-qualified pointers cannot reside in the code area as actual pointers. Conversion from the form in which `const`-qualified pointers are stored to actual pointers adds overhead to the code using `const`-qualified pointers.

ANSI/ISO rules for permissible type conversions (such as in assignment and balancing of alternatives of the ternary operator `? :`) allow `const` qualifications to be added unobtrusively which prevents the compiler from storing `const`-qualified pointers in the code area. Code compiled in strict ANSI source mode, therefore, does not store `const`-qualified pointers in the code area. Using the `-c` switch in `ucc` mode (which stores `const`-qualified pointers in the code area) refuses to compile some code that strict ANSI source mode accepts. Pointers to `const`-qualified pointers passed between functions compiled in extended ANSI source mode and functions compiled in other source modes leads to misinterpretation of data. Therefore, do not link code compiled in different source modes that pass `const`-qualified parameters with any level of indirection.



Note

The `-c` option is not available for C++.

remote

The `remote` keyword is generally not required and is only available in `compat` mode. For the 68K family of processors, if data elements go over the 64K boundary, they are placed in a `remote vsect`.

C++ Features and Restrictions

Ultra C++ imposes the following restrictions:

- Code area const pointers: the `-c` option is inapplicable in C++ mode
- `_asm` is not supported in C++ mode. If there is a need for using `_asm` in conjunction with C++, then it can be done by placing the `_asm` directives in a C file.
- Building an OS-9 module with code and data in it must be linked using the executive (`xcc`) when the source language is C++. Using the object-code linker is not sufficient, as code to call static data initializers are not generated otherwise.

Support for C++ exceptions are enabled by default in Ultra C++. For size and speed reasons, these can be disabled with the `-qnx` option; however, such a C++ program needs non-standard extensions to report language and library errors correctly. This is discussed in the section entitled, [Compiling with Exceptions Disabled](#), in [Chapter 12](#).

Object Size and Alignment

Alignment requirements of an object are specified by identifying a number that must divide the address of any object of that type.



Note

If the divisor is one, the object has no alignment restriction as one divides any integer.

The alignment requirement of a structure or union is the maximum alignment of the members. The alignment requirement of an array type is that of the type of its elements.

Aside from bit fields, the offset of a structure or union member is a multiple of its alignment requirement, hence, the above rule for the alignment requirement of structures and unions as a whole. Again, for bit fields, that constraint applies to the addressable storage units that contain them.

It remains only to specify the alignment requirements of the scalar types and to specify the integer type for a given enumeration type. Sign or lack of sign does not affect alignment so the unmodified types only are shown in the ***Ultra C/C++ Processor Guide*** in the `Language Features` sections of the processor chapters.

Compatibility with the Microware K&R C Compiler

Every effort has been made to ensure as much compatibility between the Microware K&R C compiler and Ultra C/C++ as possible. However, there are differences between the two compilers to note when compiling existing K&R source code with Ultra C/C++.

Ultra C/C++ can compile in K&R source mode in each of the executive option modes identified in the following table.

Table 2-6 Ultra C/C++ Compatibility

Mode	Option
<code>compat</code>	By default, the <code>compat</code> source mode allows for the greatest level of compatibility between the Microware K&R C compiler and Ultra C/C++. <code>compat</code> accepts code from the Microware K&R C compiler without using command line options.
<code>ucc</code>	The <code>-bc</code> option accepts code from the Microware K & R C compiler.
<code>c89</code>	The <code>-bc</code> option accepts code from the Microware K & R C compiler.



For More Information

For `fopen()` append bit information for 68K family of processors, refer to the 68K chapter in the *Ultra C/C++ Processor Guide*.

Running Compiler Makefiles

The executive and the `os9make` utility have modes allowing them to work like the Microware K&R C compiler executive and `os9make` utility. Most makefiles should work without modification. Consider the following makefile for a fictitious utility called `util` (taken from the OS-9 for 68K utility directory).

```
#
# makefile for util utility
#
OBJ      = util
ROOT     = ../..
RFILES  = util.r util1.r util2.r util3.r util4.r util5.r util6.r
RDIR     = RELS
CFLAGS  = -v=$(ROOT)/DEFS -O=2
LFLAGS  = -b -oLM=8k

$(OBJ): $(RFILES)
        cd $(RDIR); xcc $(RFILES) $(LFLAGS)

$(RFILES): utildefs.h
```

To run the makefile without modification, set the `CC` environment variable to `compat` with the command line:

```
setenv CC compat
```



Note

The `xcc` notation cannot be used for all processors. It is suggested that you use the `-tp=<x>` notation, where "x" is a reference to your target processor.

Differences Between Compatibility Source Mode and the Microware K&R C Compiler

Differences between compatibility source mode and the Microware K&R C compiler are:

- **Using Remote as a Storage Class**
- **C Keywords and Struct Member Names**
- **Error Checking**
- **Name Clashing**
- **The Executive**
- **Using csl**
- **Multiple Copies of Some Library Functions**
- **Using Libraries**
- **Using Assembly Language**
- **Standard I/O and Microware K&R C Compiler ROFs**

Using Remote as a Storage Class

Remote is accepted as a storage class in compatibility source mode. However, it is passed through the compiler with no effect. To get remote, use the `-tp` option in the compatibility option mode and pass it the `ld` target processor option.

C Keywords and Struct Member Names

The compiler does not accept C keywords for `struct` member names.

Error Checking

Ultra C/C++ has improved error checking methods over the Microware K&R C compiler. As a result, compiling source code that compiled without errors or warnings with Microware K&R C compiler may now generate errors or warnings.

Name Clashing

Because a large number of additional definitions are continually added, new names may clash in the name space with the names you previously chose. Microware attempts to avoid name clashing by creating only new names that are within the scope designated for the implementor under ANSI/ISO.

The Executive

The compiler in `compat` option mode is logically equivalent to the Microware K&R C compiler. The compilers are similar but have completely different internal structures. Makefiles and shell scripts that reference `cpp`, `cprep`, `c68`, `c68020`, or `o68` with the Microware K&R C compiler are not compatible with the compiler.

Using `cs1`

`cio` is not used by the compiler; however, all old programs continue to use `cio` until recompiled with Ultra C/C++. Therefore, `cio` remains for compatibility. All new code generated by the compiler uses `cs1`. While the main emphasis of `cio` is on I/O functions, the main emphasis of `cs1` is on ANSI/ISO functions. If special system provisions were required for `cio`, the same provisions are required for `cs1`. For example, many start-up files contain the following line:

```
link cio
```

This line should be changed to:

```
link cio cs1
```

The `cs1` module must reside in memory to link to it. Two methods of loading `cs1` to memory follow.

1. Create a new bootfile with the `cs1` module included
2. Add one of the following lines to start-up, dependent upon the hardware processor:

```
load -d /h0/MWOS/OS9/68000/CMDS/BOOTOBS/csl
```

-or -

```
load -d /h0/MWOS/OS9/68020/CMDS/BOOTOBS/csl
```

Multiple Copies of Some Library Functions

For 68K and OS-9/80x86 processors, two copies of some library functions exist in the compiler library for compatibility with the Microware K&R C compiler. [Table 2-7](#) and [Table 2-8](#) identify 68K and OS-9/80x86 functions respectively. The Microware K&R C compiler functions have the standard names. These functions accept the same parameters as they did on the Microware K&R C compiler. The functions, identified by `_ansi` preceding the function name, conform to the ANSI/ISO standard for these functions. The two versions are necessary because of the differences in functionality between the Microware K&R C compiler and the ANSI/ISO C standard.

Functions with two copies are identified by processor in the following tables.

Table 2-7 68K Processor Library Functions

Function	Compatibility	ANSI or ANSI Extended
<code>printf()</code>	•	
<code>__ansi_printf()</code>		•
<code>scanf()</code>	•	
<code>__ansi_scanf()</code>		•
<code>fopen()</code>	•	
<code>__ansi_fopen()</code>		•
<code>freopen()</code>	•	
<code>__ansi_freopen()</code>		•

Table 2-8 OS-9/80x86 Processor Library Functions

Function	Compatibility	ANSI or ANSI Extended
<code>fopen()</code>	•	
<code>__ansi_fopen()</code>		•
<code>freopen()</code>	•	
<code>__ansi_freopen()</code>		•

For all processors not specified above, **Table 2-9** identifies the two copies of some library functions existing in the compiler library for compatibility with the Microware K&R C compiler. Compiler functions have the standard names and conform to the ANSI/ISO standard. The second copy of the functions, identified by `__kandr` preceding the function name, accept the same parameters as on the Microware K&R C compiler. The two versions are necessary because of the differences in functionality between the Microware K&R C compiler and the ANSI/ISO C standard.

Table 2-9 Library Functions for All Other Processors

Function	Compatibility	ANSI or ANSI Extended
<code>fopen()</code>		•
<code>__kandr_fopen()</code>	•	
<code>freopen()</code>		•
<code>__kandr_freopen()</code>	•	

Using Libraries

Even though the latest in optimization techniques have been used, executables compiled with Ultra C/C++ may be larger than the executables for the same programs compiled with the Microware K&R C compiler. This is usually not an optimization problem but is due to some functions in the compiler library being larger and more complex than those used for the Microware K&R C compiler.

Using Assembly Language

`#asm/#endasm` and `@` may only be used for assembly language escapes in K&R source mode. Microware provides a utility called `deasm` to convert old style assembly language to new style syntax. `deasm` is documented later in this chapter.

The preprocessor exists in the front end of the compiler.

Static variables may not be accessed in assembly language with the name that might be expected. The compiler assigns unique names for these variables.

The compiler provides a clean method for including assembly level code in C source files with the `__asm()` pseudo function.



For More Information

Refer to the **Assembly Language in C Source Files** section in this chapter for additional information.

Using `deasm`

`deasm` converts the old style assembly language escapes to the new style compiler assembler escapes. It accepts input from standard input and outputs to standard out.

For example, the following file, `old.c`, contains the old style assembly language escapes:

```
#asm
junk: move.l d1,d0
      add.l #-1,d0
#endasm
@ rts
```

To change `old.c` to the new style, enter the following command at the DOS prompt:

```
deasm <old.c >new.c
```

`new.c` is as follows:

```
_asm("
junk: move.l d1,d0
      add.l #-1,d0
");
_asm(" rts");
```

Standard I/O and Microware K&R C Compiler ROFs

If you have third-party libraries (any ROF not compiled with Ultra C/C++) that deal directly with standard input, output, or error, you may **not** use the macros provided in the `stdio.h` header file. These macros do not call the functions necessary to provide compatibility. To eliminate the macros from `stdio.h`, define the preprocessor symbol

```
_NO_STDIO_MACROS
```

on the compiler executive command line.

Examples

In the first example, the program's main files are linked and the total non-remote data requirements of `cstart.r` and the object libraries to be object linked are estimated as 12K. The back end generates code as if 52K of non-remote (short access) data is available for the data. The `-p1` option is not necessary because all external data is in the non-remote area. The result is that the program's use of the data area is as optimal as the memory estimate.

```
xcc -eil file1.c file2.c file3.c file4.c -fd=files.i -tp=< >
xcc -bem=52k files.i -tp=< >
```

In this next example, the program's main files, the standard libraries, and a personal library are linked. Then, an estimate is made for the total non-remote data requirements of `cstart.r` object linked as 1K. The back end generates code as if 63K of non-remote (short access) data is available for the data it sees. `-p1` is not necessary because all external

data is in the non-remote area. The result is that the program's use of the data area is as optimal as the memory estimate. The memory estimate is more accurate and lower than the last example, so the back end sees and arranges more of the data area optimally.

```
xcc -eil file1.c file2.c file3.c file4.c -fd=files.i -j -tp=< >  
xcc -bem=63k files.i -l=mylib.1 -tp=< >
```

In the last example, the program's main files, the standard libraries, and a personal library are again linked, and the back end performs pre-linking to determine the memory requirements of `cstart.r`. The back end then generates code as is exactly needed for the program.

`-pl` is not necessary as all external data is in the non-remote area. The program's use of the data area is optimal.

```
xcc -eil file1.c file2.c file3.c file4.c -fd=files.i -j  
-l=mylib.1  
-tp=< >  
xcc -n files.i -l=mylib.1 -tp=< >
```



Note

Using the `-n` option for code that contains embedded assembly language data storage definitions may require the use of the `-bem` option if there is storage allocation in the embedded assembly.

Chapter 3: Using the Executive

The executive provides the interface between the user and the compiler and acts as the compiler control program. Through the executive, processing of source code may be controlled.

This chapter covers:

- **The Executive (xcc) Option Modes**
- **Environment Variables**
- **Include File Search Path Algorithm**
- **Library File Search Path Algorithm**
- **Predefined Macro Names for the Preprocessor**
- **Compiler Phase Codes**
- **Option Modes**
- **Library Naming Conventions**
- **Command Line Options**



The Executive (xcc) Option Modes

The cross compiler executive (xcc) operates in separate, mutually exclusive option modes defined in [Table 3-1](#).

Table 3-1 Executive Option Modes

Mode	Description
ucc	ucc Executive Option Mode. The default mode developed for Ultra C/C++. This mode enables greater control of the phases.
c89	c89 Executive Option Mode. This mode is similar to the POSIX 1003.2 compiler
compat (not available for C++)	Microware K&R C Compiler Executive Mode. This mode is compatible with the Microware K&R C compiler.

Each option mode is a complete executive in itself. Information and command line options specific to the individual option modes are provided later in this chapter.

Use the `CC` environment variable or the `-mode` command line flag to select the option mode. Refer to the next section, [Environment Variables](#).



Note

The help option, `-?`, prints help text specific to the active option mode.

Environment Variables

The compiler uses five environment variables as shown in the following tables.

Table 3-2 Environment Variables

Name	Description
CC	Selects the executive option mode
CDEF	Selects directories to search for the standard <code>#include</code> files
CLIB	Selects directories to search for the standard I-code and object code library files
MWOS	Sets the pathlist to the root of the standard MWOS file structure
TMPDIR	Selects a device or directory for temporary files

CC: Change the Executive Option Mode

The `CC` environment variable specifies the executive option mode for future compilations. There are two ways to change the executive option mode: for future compilations or for a specific compilation.

For future compilations, set the environment variable `CC`:

```
set CC=c89
```

For a specific compilation only, use the `-mode` command line flag:

```
xcc -mode=c89 file.c
```

Mode identifiers shown in [Table 3-1](#) are available for the `CC` environment variable and `-mode`. The values are not case-sensitive.

CDEF: Select Directories for Standard #include Files

The CDEF environment variable may be set to the directories in which the compiler searches for the standard `#include` files. For example, setting the CDEF environment variable as follows:

```
set CDEF=C:\DEFS\MYDEFS
```

causes the compiler to search the `C:\DEFS\MYDEFS` directory for `#include` files.

The value of CDEF may contain multiple directories separated by a semi-colon (;) pathlist delimiter character. This enables searching for `#include` files in multiple paths.

The following options add directories to be searched for `#include` files before those specified in the CDEF environment variable:

```
ucc and compat = -v[=]<dir>
```

```
c89 = -I[=| ]<dir>
```

CLIB: Select Directories for Standard I-Code and Object Code Library Files

The CLIB environment variable may be set to the directories in which the standard I-code and object code library files reside. For example, setting the CLIB environment variable as follows:

```
set CLIB=/h0/USR/ME/MYLIBS
```

causes the compiler to search the `/h0/USR/ME/MYLIB` directory for library files.

The value of CLIB may contain multiple directories separated by a semi-colon (;) pathlist delimiter character. This enables searching for library files in multiple paths.

The `-w` command line option, regardless of option mode, overrides this environment variable and the standard default library file location.

The following options add directories to be searched for libraries, before those specified in the `CLIB` environment variable.

```
c89= -L
ucc= -sl
```

MWOS: Set the Root for the Standard MWOS File Structure

MWOS sets the pathlist to the root of a standard MWOS file structure on the system where compilation is performed. In the following example, the executive looks in the `C:\PRIVATE_MWOS` directory for the subdirectories containing include and library files:

```
set MWOS=C:\PRIVATE_MWOS
```

In `ucc` mode, the same concept is accomplished using the `-mw` command line option.



Note

The MWOS environment variable is normally preconfigured for you during product installation.

TMPDIR: Select a Device or Directory for Temporary Files

The environment variable TMPDIR may be set to the device or directory the executive uses for temporary files. For example, setting the TMPDIR environment variable as follows:

```
set TMPDIR=C:\TEMP
```

causes the compiler to use the C:\TEMP directory for temporary files.

The following options override the value of TMPDIR:

```
ucc and c89 = -td
```

```
compat = -t
```

Include File Search Path Algorithm

`#include` files locations may be specified using the following command line options:

```
ucc and compat    = -v
c89               = -I
```

The compiler searches for `#include` files using the following algorithm steps.

1. Directories specified with `-v` or `-I` options are searched.
2. If the `CDEF` environment variable is set, each pathlist delimiter from the environment variable is searched in the order specified and the remaining steps are skipped.
3. If the `MWOS` environment variable is set or the `/<default_device>/MWOS` directory exists, each relevant include file directory for the target operating system from the standard `MWOS` file structure is searched and the remaining steps are skipped. The existence of the `MWOS` environment variable is checked before the existence of the directory `/<default_device>/MWOS`.
4. If both steps 2 and 3 above are unsuccessful, a single include file directory on `<default_device>` relevant to the target operating system is searched.

The algorithm is compatible with Microware K&R C compiler for systems not conforming to the standard `MWOS` file structure.

The include file directories are printed when the options to display but not execute phases (`-b` and `-h` in `c89` or `ucc` option mode, `-bp` and `-h` in `compat` option mode) are used.

Library File Search Path Algorithm

The library file path is based on the identified target processor (`-tp` option) and the host. The I-code libraries and files are host dependent due to byte ordering differences in the hardware, file object size, and alignment.

Table 3-3 Host Directory

Host	Host Directory
PC (WIndows)	HOST3

The `-w` command line option is used, regardless of option mode, to identify the location of necessary library files to the executive. The `-s1` option (`uCC` option mode) or `-L` option (`c89` option mode) specifies additional directories comprising user library files.

The executive uses the following algorithm steps to build a list of directories to search for library files.

1. The executive adds the `-s1` options (`uCC` option mode) or `-L` options (`c89` option mode) to the list in the order specified on the command line.
2. If `-w` options are specified on the command line, the executive adds the `-w` options to the list in the order specified on the command line and skips the remaining steps.
3. If the `CLIB` environment variable is set, it adds each pathlist delimiter from the environment variable in the order specified to the list. If `CLIB` is set, the remaining steps are skipped.

4. If the `MWOS` environment variable is set or the `/<default_device>/MWOS` directory exists, the executive adds the relevant library file directories for the target operating system from the standard `MWOS` file structure. The executive checks for the existence of the `MWOS` environment variable before the existence of the `/<default_device>/MWOS` directory. If `MWOS` is set or `/<default_device>/MWOS` exists, step 5 is skipped.
5. The executive adds a single list entry for the `<default_device>` library file directory relevant to the target operating system.

Once the list is constructed, the executive begins each search for a library file by checking the current data directory and then each entry in the list of library directories. If the executive cannot find or open a library, it aborts immediately with an appropriate error message.

The library file directories are printed when the options to display but not execute phases (`-b` and `-h` in `c89` or `ucc` option mode, `-bp` and `-h` in `compat` option mode) are used.

Predefined Macro Names for the Preprocessor

The macro names in [Table 3-4](#) are predefined in the preprocessor for all preprocessor targets. For processor predefined macro names, refer to the *Ultra C/C++ Processor Guide*.

Table 3-4 Predefined Macro Names

Macro Name	Description
<code>__ANSI_EXT</code>	Compiler is operating in extended ANSI source mode (value = 1)
<code>__BIG_ENDIAN</code>	Target processors using most significant byte first ordering scheme
<code>__LIL_ENDIAN</code>	Target processors using least significant byte first ordering scheme
<code>__LONGLONG_BIT</code>	Indicates the number of bits used to implement the type "long long" for the target processor. (May have a value of 32 or 64.)
<code>__MAJOR_REV</code>	Indicates the major revision number of the compiler you are using. (current value = 2)
<code>__MINOR_REV</code>	Indicates the minor revision number of the compiler you are using. (current value = 1)
<code>__OS9000</code>	Non 68K, OS-9 target systems
<code>__OS9THREAD</code>	Defined when the <code>-mt</code> option is used to enable multithreading support
<code>__OS9THREAD_UNSAFE</code>	Defined when the <code>unsafe_libs</code> option is used with the <code>-mt</code> option (<code>__OS9THREAD</code> is also defined).

Table 3-4 Predefined Macro Names (continued)

Macro Name	Description
<code>_OSK</code>	OS-9 for 68K target systems
<code>OSK</code>	OS-9 for 68K target systems (obsolete: appears only when compiling in the <code>compat</code> source mode; use <code>_OSK</code>).
<code>__SPACE_FACTOR</code>	The value of the space weight option (<code>-s</code> in <code>ucc</code> option mode and <code>-m</code> in <code>c89</code> option mode)
<code>__STDC__</code>	Compiler is operating in ANSI C-conforming source mode (value = 1)
<code>__TIME_FACTOR</code>	The value of the time weight option (<code>-t</code> in <code>ucc</code> option mode and <code>-n</code> in <code>c89</code> option mode)
<code>_UCC</code>	The Ultra C/C++ compiler is being used (value = 1)
<code>__cplusplus</code>	Compiler is operating in C++ source mode
<code>__NO_EXCEPTIONS</code>	C++ exception handling has been disabled

Target names identify the compiler when writing machine-independent and operating system-independent programs.



Note

Refer to the *Ultra C/C++ Processor Guide* for information about the relationship between the target processor and the preprocessor macros.

Refer to the `-tp` option described in the *Ultra C/C++ Processor Guide* for related information.

Compiler Phase Codes

Each compiler phase is specified by a two-character code. In the following command line option sections, <phase> specifies one of the codes identified in [Table 3-5](#). Not all phases are valid for all command line options that allow specification of a phase code.



For More Information

For a description of command line options valid by phase, refer to [Chapter 5: Compiler Phase Options](#).

Table 3-5 Compiler Phase Codes

Code	Phase
fe	Front end
il	I-code linker
io	I-code optimizer
be	Back end
ao	Assembly optimizer
as	Assembler
pl	Prelinker
ol	Object code linker



Note

Phase specification is not valid in `compat` option mode.



Note

The help option, `xcc -<compiler phase code>?`, prints help text specific to the compiler phase.

Option Modes

Three mutually exclusive executives exist in Ultra C/C++ and are named:

- **ucc Option Mode**
- **c89 Option Mode**
- **compat Option Mode**

Executive option modes are described in the following subsections.

ucc Option Mode

In `ucc` option mode, the compiler recognizes many command line options that modify compilation. These options are not case sensitive and all options are parsed before compilation begins. Consequently, options may be placed anywhere on the command line. Options may be grouped (for example, `-bh`) except where an option specifies an argument (for example, `-f=<path>`).

`ucc` option mode defaults to the extended ANSI source mode and automatically links with the `os_lib.l` and `clib.l` libraries for C and `os_lib.l`, `clib.l`, and `cplib.l/cplibnx.l` for C++.

File Name Extensions

Table 3-6 identifies the file name extension conventions used in `ucc` option mode.

Table 3-6 `ucc` File Type/Extension

File Type	Extensions
C/Preprocessed Source	.c, .pp
C++	.cpp, .cxx, .c

Table 3-6 ucc File Type/Extension (continued)

File Type	Extensions
I-code	.i
Back End Output	.o
Assembly	.a
ROF	.r, .l

**Note**

Note the uniqueness of the file types for the various extensions across executive modes — there are clashes with .a and .o files produced by other modes.

c89 Option Mode

In c89 option mode, the compiler recognizes many command line options that modify compilation. All options are parsed before compilation begins. Consequently, options may be placed anywhere on the command line. Options may be grouped (for example, `-sr`) except where an option specifies an argument (for example, `-f <path>`).

c89 option mode defaults to the ANSI source mode and automatically links with the `os_lib.l` and `clib.l` libraries for C and `os_lib.l`, `clib.l`, and `cplib.l/cplibnx.l` for C++.

Behavior

To comply with the POSIX 1003.2, `c89` does not conform to all Microware conventions (for example, argument parsing, file name extensions, default output file naming).

Option Parsing

Extensions to the POSIX draft are included. However, POSIX may eventually clash with Microware's choice of option names. The standard POSIX options are:

`-c`, `-g`, `-s`, `-o`, `-D`, `-E`, `-I`, `-L`, `-l`, `-O`, `-U`, `-P`, `-S`, and `-W`

All other options are Microware extensions to the draft and may change in subsequent releases.

`c89` mode has a flexible option parser. The following four forms of an example option (`-o`) and its argument (`arg`) are accepted and equivalent:

```
-o arg
-oarg
-o=arg
"-o arg"
```

File Name Extensions

Table 3-7 identifies the file name extension conventions used in `c89` option mode:

Table 3-7 `c89` File Type/Extension

File Type	Extensions
C/Preprocessed Source	<code>.c</code> , <code>.i</code>
C++	<code>.cpp</code> , <code>.cxxx</code> , <code>.c</code>
I-code	<code>.ic</code>

Table 3-7 c89 File Type/Extension (continued)

File Type	Extensions
Back End Output	.be
Assembly	.s
ROF	.o, .r, .a, .l

**Note**

Note the uniqueness of the file types for the various extensions across executive modes - there are clashes with .a, .o, and .i files produced by other modes.

Executable Output File

As required by the POSIX draft, if `-o` option is not used, the default name for the executable object code file in `c89` option mode is `a.out`. This does not imply that the file is of any standard format other than that of an OS-9 module.

**Note**

Use `-o` to produce a different executable rather than renaming the file after creation.

The permission bits on the output file are such that the user running the executive can read, write, and execute the file. That is, an implied `umask` disables all public (as well as group, in OS-9) permissions.

Library Specification with -l

The POSIX draft specifies that the operand to `-l`, library, is used to build a name in the form:

```
lib<library>.a
```

The executive searches for a file with the proper name, `liblibrary.a`, in the directories specified with the `-L` option.

For example, if the following command line is entered, the executive searches the `/h0/PROJ/LIB` directory for the `libbeta.a` library file:

```
xcc -mode=c89 example.c -l beta -L /h0/PROJ/LIB
```

If a directory with the `-L` option is not specified, the executive, at a minimum, searches the current data directory and the default library directories.

If a file is not found with this name, the name is assumed to be a direct pathlist to a library. Libraries specified with `-L` are searched at object code link time in the order specified on the command line.

To use libraries shipped with Ultra C/C++, perform one of the following:

- Rename the libraries adhering to the naming convention
`lib<library>.a`
- Copy the libraries into files adhering to the naming convention
`lib<library>.a`

Four arguments to `-l` are reserved: `c`, `m`, `l`, and `y`. They are parsed but ignored.

compat Option Mode

In `compat` option mode, the compiler recognizes many command line options that modify compilation. Options are not case significant and may be placed anywhere on the command line as all options are parsed before compilation. Options may be grouped (for example, `-sr`) except where an option specifies an argument (for example, `-f=<path>`).

`compat` option mode defaults to the K & R source mode and automatically links with the `sys_clib.l`, `os_lib.l`, and `clib.l` libraries.



Note

`compat` option mode is available for compatibility with the Microware K&R C compiler. It enables processing of old makefiles created with the Microware K&R C compiler to work. For new makefiles, use either the `ucc` or `c89` option mode.

Library Naming Conventions

For backward compatibility and flexibility, there are two forms of many libraries. This enables you to choose between a multiple thread or a single threaded strategy. The multiple thread libraries have locking around vulnerable data to protect thread collisions. The single thread library does not have this locking in place. The naming convention for these two libraries is as follows:

```
mt_<library name>    * multi thread safe
<library_name>      * single threading with no locks
```



Note

If the `-mt` option is not used in the command line, the executive automatically searches for the non `mt_` library name.

Command Line Options



Note

A compiler phase code can precede many command line options in `ucc` and `c89` mode, allowing passing of an option to a specific compiler phase.



For More Information

Refer to the `-<phase> [=] <opt>` command for instructions on prepending a phase code in `ucc` mode. Refer to the `-W[<pass>] , <arg1> [, <arg2> . . .]` command for instructions on prepending a phase code in `c89` mode.

Chapter 5: Compiler Phase Options identifies phase codes and command line options valid for each phase.

Command line options, identified by executive option mode, are defined in [Table 3-8](#).

Table 3-8 Command Line Options

Option	uCC	c89	compat	Description
-?	•	•	•	<ul style="list-style-type: none"> Lists all command line options for a specific mode.
-<phase>? Get Option Information about Specific Phase	•			<p>Provides help on command line options for a specific phase. <phase> is one of the phases listed in Chapter 5: Compiler Phase Options.</p> <p>For example, use the following command to get command line option information from the front end and the assembler:</p> <pre>xcc -fe? -as?</pre> <p>When this option is used, compilation does not occur.</p>
-<phase> [=] <opt> Pass Option to Specified Phase	•			<p>Passes an option to <phase>. <phase> is identified and described in Chapter 5: Compiler Phase Options.</p> <p>For example, pass -s and -g to the object code linker with the command line:</p> <pre>xcc test.c -o1S -olg</pre>
-A	•			<p>Allow C++ style comments to be used in C code.</p> <p>This option is only available while in extended ANSI.</p>

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-a Generate Assembly Output				<ul style="list-style-type: none"> Leaves the output as assembler code in a file with the .a suffix.
-a[[=]<type>] Compile in Strict ANSI Source Mode				<ul style="list-style-type: none"> Specify -a when the code to compile requires checking for strict ANSI/ISO-compliance. <type> can be:
-a[[=]<type> Compile in Strict ANSI Source Mode				<ul style="list-style-type: none"> warn = Issue warnings for violations err = Issue errors for violations
-ae Compile in ANSI Extended Source Mode				<ul style="list-style-type: none"> Use -ae when the source files to process contain elements of non-ANSI Microware extensions.
-b Verbose Command Line Output				<ul style="list-style-type: none"> When -b is used, the compiler prints the command lines executed to accomplish each phase. When this option is used with -h, the paths to the library files and include files are printed.
-bc Compile in K & R Source Mode				<ul style="list-style-type: none"> -bc is recommended when source files contain elements specific to the Microware K&R C compiler. Refer to the Ultra C/C++ Processor Guide for more information on compatibility between the two compilers.
-bg Set Sticky Bit in Module Header				<ul style="list-style-type: none"> Set the sticky bit in the module header to cause the module to remain in memory, even if the link count becomes zero.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-bp Print Arguments Passed to Phases				<ul style="list-style-type: none"> • Print the arguments passed to each compiler phase and an exit status message. Enables determination of arguments that the executive passes to each phase.
-C Include C Source Code as Comments in Generated Assembly Files				<ul style="list-style-type: none"> • Include C source code as comments in generated assembly files.
-c Const Qualified Pointers in Code Area				<ul style="list-style-type: none"> • Allow the addition of constraints on <code>const</code> qualified pointers enabling storage of the pointers in the code area. <code>const</code> qualified pointers are treated differently than normal pointers. Therefore, their values may never be assigned to or interpreted as normal pointers. This option is not available for C++ code. <p>Refer to Chapter 2: Compiling for more information on using <code>const</code> with the compiler.</p>
-c Suppress Linking ROF Modules into Executable Programs				<ul style="list-style-type: none"> • Output is left in files with a <code>.o</code> suffix. This produces a <code>.o</code> file for each <code>.c</code> or <code>.i</code> file given on the command line.
-c Print Source Code as Comments with Assembler Code				<ul style="list-style-type: none"> • <code>-c</code> is most useful with the <code>-a</code> option.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-co Include C Source Code as Comments in Generated Assembly Files	•			Include C source code as comments in generated assembly files.
-cq	•			Allow C++ style comments to be used in C code.
-cs[=]<root psect> Specify Alternate Root psect	•			<ul style="list-style-type: none"> The path name is considered relative to the current data directory. By default, the root psect is <code>ansi_cstart.r</code> in <code>uCC</code> mode and <code>cstart.r</code> in <code>compat</code> mode
-cx[[=]<file>]	•			Extended cross reference information.
-cw Enable Warnings	•			Enable warnings in various compiler phases. Warnings may be generated as a result of the following conditions: <ul style="list-style-type: none"> Function defined or called without a prototype. A potential uninitialized variable exists. A function defined as returning a value did not return a value.

Table 3-8 Command Line Options (continued)

Option	ucc	c89	compat	Description
-D[=]<name>[=<value>] Define Name and Value for Preprocessor and Assembler		•		Define a name and optionally a value for the preprocessor and the assembler. This is useful when different versions of a program are maintained in one source file and differentiated
-d[=]<name>[=<value>] Define Name and Value for Preprocessor and Assembler	•		•	through the <code>#if defined()</code> or <code>#if !defined()</code> preprocessor directives. If <code><name></code> is used as a macro for the preprocessor to expand, 1 (one) is the expanded value unless an expansion string is specified using the form <code>-D <name>=<value></code> in c89 mode and <code>-d <name>=<value></code> in ucc and compat modes.
-E Preprocess to Standard Out			•	Preprocesses all command line .c files to standard output.
-e[=]<num> Set Edition Number			•	Set the edition number constant byte to the specified number. This option is an OS-9 convention for memory modules.

Table 3-8 Command Line Options (continued)

Option	ucc	c89	compat	Description
-e[=]<phase>[[=]<dir>] Specify Endpoint of Compilation Process	•			<phase> is identified and described in Chapter 5: Compiler Phase Options . <dir> specifies where to place the output file.
-e[=]<phase>[[=]<dir>] Specify Endpoint of Compilation Process		•		<p>NOTE: <code>o1</code> (object code linker) is not a valid phase for <code>-e</code>.</p> <p>To create a <code>.i</code> file, stop the compile at the front end, I-code linker, or I-code optimizer.</p> <p>ucc mode example:</p> <pre>ucc prog1.c -eil</pre> <p>c89 mode example:</p> <pre>ucc prog1.c -e il</pre> <p>To create a <code>.r</code> file, stop the compile at the assembler. The following examples also place the file in the <code>RELS</code> directory.</p> <p>ucc mode example:</p> <pre>ucc prog1.c -eas=RELS</pre> <p>c89 mode example:</p> <pre>ucc prog1.c -e as=RELS</pre>

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-f [=]<path> Override Output File Naming Conventions	•			<ul style="list-style-type: none"> The last element of <path> specifies the name of the output file. The module name is the same as the file name unless the linker <code>-n=<name></code> option is used. <p>In c89 mode, the executive <code>-n</code> option may also be used to alter the module name.</p> <p>On OS-9 systems, if <path> is a relative pathlist, it is relative to the current execution directory.</p>
-f [=]<path> Override Output File Naming Conventions			•	<p><path> specifies the pathlist and name of the output file. This name is used for the output of the final compilation phase specified with <code>-e</code> or the object code link phase if <code>-e</code> not is specified. If <code>-e</code> is used with a directory for the output, the name given with <code>-f</code> is concatenated to that directory name.</p> <p>On OS-9 systems, if <path> is a relative pathlist, it is relative to the current data directory.</p>
-fd [=]<path> Override Output File Naming Conventions			•	<ul style="list-style-type: none"> The last element of <path> specifies the name of the output file. The module name is the same as the file name unless the <code>-n=<name></code> option is used. <p>On OS-9 systems, if <path> is a relative pathlist, it is relative to the current data directory.</p>

Table 3-8 Command Line Options (continued)

Option	ucc	c89	compat	Description
-fd[=]<path> Override Output File Naming Conventions	•			<p><path> specifies the pathlist and name of the output file. This name is used for the output of the final compilation phase specified with <code>-e</code> or the object code link phase if <code>-e</code> is not specified. If <code>-e</code> is used with a directory for the output, the name given with <code>-fd</code> is concatenated to that directory name.</p> <p>On OS-9 systems, if <path> is a relative pathlist, it is relative to the current data directory.</p>
-g Output Symbol Modules for Debugging	•	•	•	<p>Cause the linker to output two symbol modules for use by Microware debuggers. The modules have the same name as the output file with <code>.stb</code> and <code>.dbg</code> appended respectively. If an <code>STB</code> directory exists in the target output directory, the symbol module is placed there, otherwise it is placed in the same directory as the output file.</p> <p>NOTE: The <code>-g</code> option overrides the <code>-o</code> option in <code>ucc</code> and <code>compat</code> modes and the <code>-0</code> option in <code>c89</code> mode.</p>

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
<p><code>-h</code> Suppress Phase Execution</p>	•	•	•	<p>Specify <code>-h</code> (with <code>-b</code> in <code>uCC</code> and <code>c89</code> modes or <code>-bp</code> in <code>compat</code> mode) to examine the compiler phases and command line options. Compilation does not occur.</p> <p>WARNING: Attempting to use the output of <code>xCC -bph</code> as a procedure (script) may be unsuccessful. The unique temporary file creation mechanism is circumvented when phases do not execute.</p>
<p><code>-I[=] <dir></code> Specify Additional Directory to Search for Preprocessor <code>#include</code> Files</p>			•	<p>File names within quotes are searched for in the current directory. File names within angle brackets (<code>< ></code>) are searched for in the specified directory.</p> <p><code>-I</code> may appear more than once. In this case, each directory is searched in the order specified on the command line.</p> <p>For More Information: The algorithm used to search for <code>#include</code> files is covered in the Include File Search Path Algorithm section earlier in this chapter.</p>
<p><code>-i</code> Link Program with C Shared Library</p>	•	•	•	<p>A <code>cs1</code> (C shared library) module processes references to selected C functions.</p>

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
<p>-j</p> <p>Include I-Code Versions of Standard Library on I-Code Link Line</p>	•	•		<p>Cause the I-code linker to include the I-code versions of the standard library on the I-code link line. This enables the optimizer to perform optimizations involving library functions.</p> <p>WARNING: Object code linking of ROFs that have been I-code linked with the standard libraries causes multiple occurrences of symbol defines.</p>
<p>-j</p> <p>Prevent Linker from Creating Jumptable</p>				<ul style="list-style-type: none"> Prevent the linker from creating a jumtable.
<p>-k</p> <p>Suppress Inclusion of Default Libraries in Object Code Link Phase</p>	•	•		<p>Suppress inclusion of default libraries in the object code link phase.</p>
<p>-k[=<n>[w l][cd cl] [f]</p> <p>Define Target Processor</p>	•	•		<ul style="list-style-type: none"> Target Processor -0=68000, 2=68020, and so forth. data area offsets: w=word, l=long word code area offsets: cw=word, cl=long word floating point: f=68881 (68020 only)

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
<p><code>-L[=<i>dir</i>]<dir></code> Specify Directory Containing User Library Files</p>			•	<p><code>-L</code> is useful when used with <code>-l</code>. The order of <code>-L</code> options is significant as directories are searched in the order specified.</p> <p>NOTE: The algorithm used to search for library files is covered in the Library File Search Path Algorithm section earlier in this chapter.</p>
<p><code>-l[=<i>lib</i>]<lib></code> Include <code><lib></code> in Object Code Link Phase</p>			•	<p>Specify <code>-l</code> to link with additional user libraries. User libraries are added to the link phase in the order specified and before the standard libraries. For POSIX conformance, <code>a</code>, <code>m</code>, <code>y</code>, and <code>l</code> are valid libraries but have no effect.</p>

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
<p><code>-l[=]<path></code> Specify Library File for Search</p>	•			<ul style="list-style-type: none"> Specify a library file for the linker to search before searching the standard library and systems interface library. If the directory containing the library was previously provided (via <code>-w</code>, <code>CLIB</code>, or <code>MWOS</code>) then only the file name for the library must be specified. For example (assuming <code>/dd/MWOS</code> is available): <pre>xcc test.c -l=termlib.l</pre> Links with the object code library <code>termlib.l</code> from the standard <code>MWOS</code> file structure. <p>NOTE: The algorithm used to search for library files is covered in the Library File Search Path Algorithm section earlier in this chapter.</p>
<p><code>-lo[=]<linker options></code> Pass Specified Options to Linker</p>				<ul style="list-style-type: none"> Only white space is recognized as a delimiter; the linker does not support options that require quoted strings. <p>NOTE: Wild card characters (<code>?</code> or <code>*</code>) are disallowed in the linker options.</p>

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-M[=] <phase> [=] <num> [K k] Allocate Additional Stack Space to Specific Phase		•		<phase> is identified and described in Chapter 5: Compiler Phase Options . <num> is in terms of kilobytes.
-m[=] <phase> [=] <num> [K k] Allocate Additional Stack Space to Specific Phase		•		This option should be unnecessary under normal circumstances. For example the following commands would run the I-code optimizer with an additional 10K and the object code linker with an additional 20K of stack space:
-m[=] <num> Weight Given to Space Considerations		•		-m specifies the importance of code size for the output file. If this number is larger than that on the -n option, time is considered more important than space. The default is 1. The minimum value is -0.
-m[=] <mem size> [K] Allocate Additional Size for Program Stack			•	Instruct the linker to allocate an additional <mem size> (in kilobytes) for the program stack.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-mt [[= <i><op></i> [, <i><op></i> . . .]] Specify multi-threading options	•	•		<p>This switch is necessary when compiling or linking multi-threaded code. The following options alter the default methods of library selection and code compatibility checking.</p> <p><i><op></i> may be any of the following options:</p> <p><i>enable</i> = enable multi-threading support (default)</p> <p><i>none</i> = disable multi-threading support</p> <p>The Library selection options govern how the compiler searches for the libraries. You do not need to specify linking with multi-threaded libraries—the compiler will find them.</p> <p><i>safelibs</i> = thread-safe libraries w/ fallback (default)</p> <p><i>strictlibs</i> = thread-safe libraries only</p> <p><i>unsafelibs</i> = thread-unsafe libraries</p> <p>The Code compatibility options govern how the compiler handles the linking of incompatible code.</p> <p><i>errlink</i> = error given incompatible code (default)</p> <p><i>warnlink</i> = warn given incompatible code</p> <p><i>quietlink</i> = allow mixing of safe and unsafe code</p>

Table 3-8 Command Line Options (continued)

Option	ucc	c89	compat	Description
-mw[=]<dir> Specify Location of Standard MWOS	•			Specifies the location of a standard MWOS directory structure to use for <code>#include</code> and compiler library files. This option overrides the MWOS environment variable and <code>/<default_device>/MWOS</code> directory.
-mw[=]<dir> Specify Location of Standard MWOS		•		
-N Perform Data Area Layout		•		Causes the back end to see the object linker files enabling the back end to generate code knowing what the final data area looks like.
-n Perform Data Area Layout	•			
-n[=]<name> Specify Output Module's Name			•	Enables specification of an output module name.
-n[=]<num> Weight Given to Time Considerations			•	<code>-n</code> specifies the importance of code size for the output file. If this number is larger than that on the <code>-m</code> option, space is considered more important than time. The default is 1.
-nl Prevent Use of Default Libraries during Linking			•	<code>-nl</code> precludes use of default libraries during linking.
-nv Force Compiler to Suppress vsect Directives			•	Forces the compiler to suppress <code>vsect</code> directives from the code it generates. Use <code>-nv</code> to make OS-9 descriptors.
				NOTE: <code>-nv</code> is valid only for OS-9 targets.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
<p>-O "?" Print Help Information about Optimization Levels</p>			•	<p>The ? must be placed between quotes to prevent the shell from treating it as a wild card.</p>
<p>-o? Print Help Information about Optimization Levels</p>			•	<p>Prints help information about optimization levels.</p>
<p>-O[[=] [<num>]] Optimize</p>			•	<p>Enables maximum optimization. The optional argument specifies the optimization level. <num> should be in the range 0 (off) to 7 (maximum).</p> <p>NOTE: The -g option overrides the -o option.</p>
<p>-o[[=] <num>] Set Optimization Level</p>			•	<p>Generally, the more optimization, the faster the code execution. Refer to the -t and -s options for further optimization modification. <num> should be in the range of 0 (off) to 7 (maximum). Sets to 0 in <num> not specified.</p> <p>NOTE: The -g option overrides the -o option.</p>

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-o[[=] <level>] Set Optimization Level				<ul style="list-style-type: none"> The optimizer shortens object code and increases speed. This is recommended for production versions of debugged programs. <level> accepts the following values: <ul style="list-style-type: none"> 0 = No optimization 1 = Assembly language optimizations; I-code optimizations, except Common Subexpression Elimination (CSE) 2 = Assembly language optimizations and all I-code optimizations <p>If -o is not declared, level 1 optimization is performed. If -o is specified but the level option of the statement is not, optimization is not performed.</p> <p>NOTE: The -g option overrides the -o option.</p>
-o[=]<path> Override Object Code Linker Output File Naming Conventions				<ul style="list-style-type: none"> The last element of <path> specifies the name of the output file. In OS-9 systems, if <path> is a relative pathlist relative to the current execution directory. If -o is not specified on a command line, the file and module names are a.out.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-P Preprocess to File		•		Preprocess and send the results to standard output—this output will include #line directives, so that a later program can read and honor those directives.
-PC		•		Causes comments to be left in the output; normally they are replaced with white space, to simplify the parsing job of programs that read the preprocessed output.
-PO		•		Leaves #line directives in the output, but emits them in the style of the old "Reiser" C preprocessor that came with the Bell Labs pcc C compiler, for old programs that expect their preprocessed output to have that format for #line directives.
-PP		•		Causes #line directives to be omitted from the output; this simplifies the parsing for programs that read the preprocessed output, but may lead to confusion if the program emits error messages with line number and file name information.
-PH		•		Generates a list of included files, but no preprocessed output.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-PM		•		Generates a list of dependencies (the same as included files, but in a format that os9make can understand), but no preprocessed output.
-p[<mode>][= <dir>] Preprocess to File				<p>Use <code>-p</code> to stop compiling after the preprocessing stage. The result of preprocessing is written to standard output or to a file (with the extension <code>.pp</code>) in the directory, if given.</p> <p><mode> accepts the following:</p> <p><code>C</code> = Keeps comments in preprocessed code</p> <p><code>E</code> = Preprocess to standard output (default)</p> <p><code>H</code> = Generates a list of include files</p> <p><code>M</code> = Generates a list of dependencies</p> <p><code>O</code> = Uses old Reiser format for line information</p> <p><code>P</code> = Preprocess without line directives</p>
-p? Lists Possible Modes for -p		•		Prints a list of possible modes for <code>-p</code> option.
-p [=] <root psect> Specify Alternate Root psect			•	The path name is considered relative to the current data directory. By default, the root psect is <code>ansi_cstart.r</code> .

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-q Specify Quiet Mode				<ul style="list-style-type: none"> The executive does not announce internal steps as they occur. Only error messages, if any, are displayed.
-qb Specify C++ Compatibility with cfront 2.1	•	•		C++ source code only.
-qc Specify C++ Compatibility with cfront 3.0	•	•		C++ source code only.
-qnx	•	•		Disable exceptions; no exception processing code is linked in. In addition, the macro <code>__NO_EXCEPTIONS</code> gets defined. Refer to the C++ Features and Restrictions in Chapter 2 and the Compiling with Exceptions Disabled section in Chapter 12.
-qp Compile or Link in C++ Mode	•	•		Use <code>-qp</code> when all <code>.c</code> files on the command line are to be treated as C++ source files or when all <code>.x</code> files on the command line are to be treated as ROFs generated from C++ sources. Use of <code>-qp</code> when linking ROFs causes the linker to link C++ libraries and initialization code.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-qt [=] <mode> Specify Instantiation Mode	•	•		C++ source code only. <mode> is one of: none = No entities (default) used = Only entities used all = All entities local = Only used entities as static functions NOTE: Refer to the Instantiation Modes section in Chapter 12: Language Features , for information on the -qt option.
-r Stop Generating Stack- Checking Code	•	•		Use -r only when the application is extremely time critical and use of the stack by compiler generated code is fully understood.
-r[[=] <dir>] Suppress Linking Library Modules into Executable Programs				• Output is left in files with a .r suffix. If -r=<dir>, the .r files are placed in <dir>.
-S Suppress Assembly Phase				• Leaves the output as assembler code in a file with the .s suffix. Specify -s to generate a .s file for each command line .c or .i file.
-s Suppress Symbolic Debugging Information				• Specify -s if source level debugging is unnecessary. If both -g and -s are specified on the same command line, the right-most -g or -s takes precedence.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-s Stop Generating Stack- Checking Code				<ul style="list-style-type: none"> Use -s only when the application is extremely time critical and use of the stack by compiler generated code is fully understood.
-s [=] <num> Weight Given to Space Considerations				<ul style="list-style-type: none"> Specify the importance of code size for the output file. If this number is larger than that specified for the -t option, space is considered more important than time. The default is 1. The minimum value is 0.
-sl [=] <dir> Specify Additional Directory Containing Library Files				<ul style="list-style-type: none"> Informs the linker to search for library files in the directory specified by <dir>. This option may occur more than once. <p>NOTE: The algorithm used to search for library files is covered in the Library File Search Path Algorithm section earlier in this chapter.</p>
-t? List Target Processor- Specific Options				<ul style="list-style-type: none"> Lists target specific options.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-t [=]<dir> Place Temporary Files in Specified Directory				<ul style="list-style-type: none"> The executive places the temporary files used by any compiler phase in the specified directory. If the device containing the directory is a RAM disk device (for example, -t=B:), compile time is drastically reduced. <p>NOTE: See the TMPDIR: Select a Device or Directory for Temporary Files section in this chapter for more information about temporary file locations.</p>
-t [=]<num> Weight Given to Time Considerations			•	Specify the importance of code size for the output file. If this number is larger than that on specified for the -s option, time is considered more important than space. The default is 1. The minimum value is 0.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-td [=]<dir> Specify Directory for Intermediate Step Files	•			The executive uses a different directory/device for its temporary files.
-td [=]<dir> Specify Directory for Intermediate Step Files		•		Examples enabling a RAM disk: uCC mode: <code>xcc -td=B: test.c</code> c89 mode: <code>xcc -td B: test.c</code>
-to [=]<name> Specify Target Operating System by Name	•			NOTE: See the TMPDIR: Select a Device or Directory for Temporary Files section in this chapter for more information about temporary file locations.
-to [=]<name> Specify Target Operating System by Name			•	Valid <name> values are: <code>osk = 68K</code> <code>os9k = OS-9</code> <code>OS9000 = OS-9</code>
-tp [=]<n>[...] Specify Target Processor and Target Processor Sub-Options	•			Refer to the Executive -tp Option section in the Ultra C/C++ Processor Guide for specific target processors and sub-options.
-tp [=]<n>[...] Specify Target Processor and Target Processor Sub-Options			•	

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-u[=] <name> Undefine Previously Defined Preprocessor Macro Names	•			• The Predefined Macro Names for the Preprocessor are listed earlier in this chapter.
-U[=] <name> Undefine Previously Defined Preprocessor Macro Names		•		
-v Enable Warnings		•		Enable warnings in various compiler phases. Warnings may be generated as a result of the following conditions: <ul style="list-style-type: none"> • function defined or called without a prototype • potential uninitialized variable • a function defined as returning a value did not return a value
-v[=] <dir> Specify Additional Directory to Search for Preprocessor #include Files	•			• It is assumed that #include file names within quotes are located in the current directory and then in the specified directory. #include file names within angle brackets (< >) are searched for in the specified directory. -v may be specified more than once. In this case, each directory is searched in the order provided on the command line.
				NOTE: The algorithm used to search for #include files is covered in the Include File Search Path Algorithm section earlier in this chapter.

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-W[<pass>], <arg1>[, <arg2> ...] Pass Arguments to Specific Phases of Compilation Process		•		<p><pass> codes are available in c89 mode. The following codes may be prepended to c89 command line options to cause execution of the command in a particular phase specified by the code.</p> <p><pass> codes are:</p> <ul style="list-style-type: none"> p, 0 = Preprocessor/compiler i = I-code linker 2 = I-code optimizer b = I-code to assembly translator o = Assembly optimizer a = Assembler q = Prelinker l = Object code linker <p>For example, pass -S and -g to the object code linker with the command line:</p> <pre>xcc -o test test.c -Wl, -S, -g</pre> <p>For More Information, refer to Chapter 5: Compiler Phase Options, to determine command line options passable to a particular phase.</p>

Table 3-8 Command Line Options (continued)

Option	uCC	c89	compat	Description
-w[=]<dir> Specify Directory Containing Default Library Files	•			<ul style="list-style-type: none"> Specify the directory containing the default library files (such as <code>cstart.r</code> and <code>clib.l</code>). <code>-w</code> is useful when the library directory resides on a remote file server or custom versions of such files are used.
-w[=]<dir> Specify Directory Containing Default Library Files		•		<p>NOTES:</p> <p>This option may be used more than once on a command line.</p> <p>The algorithm used to search for library files is covered in the Library File Search Path Algorithm section earlier in this chapter.</p>
-x Generate Trap Instructions to Access Floating Point Math Routines				<ul style="list-style-type: none"> <code>-x</code> is an accepted but obsolete option and does not perform a function.
-X		•		<ul style="list-style-type: none"> Generate extended cross reference information

Table 3-8 Command Line Options (continued)

Option	ucc	c89	compat	Description
<code>-x [=] <phase> [<phase>]</code> Specify Phases to Skip	•			Eliminates one (or more) compiler phases. Phases are:
<code>-x [=] <phase> [<phase>]</code> Specify Phases to Skip		•		<code>il</code> = I-code linker <code>io</code> = I-code optimizer <code>ao</code> = Assembly optimizer
Examples:				
ucc mode:				
<code>xcc -x=ioao test.c</code>				
c89 mode:				
<code>xcc -x ioao test.c</code>				
<code>-y [=] <file></code> Specify I-Code Library File for Search	•			Specify an I-code library file for the I-code linker to search before searching the standard libraries. The libraries given with <code>-y</code> are placed after the user-specified options, therefore libraries specified with <code>-wi,-l</code> in <code>c89</code> mode or <code>-ill</code> in <code>ucc</code> mode are searched before those specified with <code>-y</code> .
<code>-y [=] <file></code> Specify I-Code Library File for Search		•		NOTE: The algorithm used to search for library files is covered in the Library File Search Path Algorithm section earlier in this chapter
<code>-z [[=] <file>]</code> Read Options and Parameters from File	•			• If <code><file></code> is not specified, the options and parameters are read from standard input.
<code>-z [[=] [<file>]]</code> Read Options and Parameters from File		•		

Chapter 4: Example Compilations

Examples for the following compilation types are provided in this chapter.

- **Single Source File Program**
- **Single Source File Program I-Code Linked with I-Code Libraries**
- **Multiple Source File Program I-Code Linked**
- **Multiple Source File Program Object Code Linked**
- **Multiple Source File Program I-Code Linked with Makefile**
- **Multiple Source File Program Object Code Linked with Makefile**
- **Multiple Source File Program I-Code Linked into Segments with Makefile**
- **Multiple Source File Non-Program I-Code Linked with Makefile**
- **Multiple Source File Object Library Creation with Makefile**
- **Multiple Source File I-Code Library Creation with Makefile**



Single Source File Program

The following command line compiles the source file `test.c` to an ROF and object code links the ROF with the default libraries. This generates a program module called `test` (the basename of the source file).

```
xcc test.c
```

Single Source File Program I-Code Linked with I-Code Libraries

The following command line compiles the source file `test.c` to an I-code file and I-code links it with the I-code libraries. The resulting file is compiled to an ROF and linked with the object code libraries. Refer to [Chapter 1: Overview](#), for more information on the advantages of I-code linking.

```
xcc test.c -j
```

Multiple Source File Program I-Code Linked

The following command line compiles `file1.c` and `file2.c` to their I-code forms and I-code links them together. If the `-j` command line option (include I-code versions of standard library on I-code link line) is used, these files are I-code linked together with the I-code libraries. This I-code file is compiled to its ROF form and object linked with the libraries to create a program called `utility`.

```
xcc file1.c file2.c -f=utility
```

Multiple Source File Program Object Code Linked

The following command line compiles `file1.c` and `file2.c` to their ROF forms and object code links them with the libraries and creates a program called `utility`. The `-x [=] <phase> [<phase>]` command line option (specify phases to skip) was used to suppress the default behavior of I-code linking multiple input files.

```
xcc file1.c file2.c -x=il -f=utility
```

Multiple Source File Program I-Code Linked with Makefile

In this example, an application is compiled by I-code linking the various source files. Advantages of I-code linking an application with the I-code versions of the standard libraries compared to object code linking are identified following.

- Branch Size Optimization

All functions are present in one I-code file, which is translated to one assembly language file. Therefore, the assembler recognizes the distance of branch instructions and minimizes the instruction length resulting in reduced code size. When object code linking, the compiler handles branches to external targets as though they span large distances and increase the instruction length.

- Interprocedural Optimization

The I-code optimizer recognizes the whole program. Therefore, it may perform optimizations otherwise impossible if each function was viewed independently.

- Data Area Layout

All parts of the program are present. Therefore, the back end arranges the data area such that the most frequently accessed global data items are placed in the area quickest to access.

The makefile used in I-code linking is similar to that used for object code linking except that I-code files rather than ROFs are compiled. The makefile compiles each source file into an optimized I-code file. The optimized I-code files are I-code linked with each other and with the I-code versions of the standard libraries. This file is once again passed through the I-code optimizer (for function inlining and other optimizations) and the remainder of the file is optimized to create an executable module.



Note

The makefile terminates commands at the end of the line, unless the line ends with a backslash (\).

CAUTION: If there is not supposed to be a space in the line, indentations will cause errors.

The makefile used for this example follows:

```
#
# makefile for Robot Controller
#
-o

TEMP    = C:\TEMP           # set temp directory
CC      = xcc               # compile command
OBJ     = robot             # object to be returned
IDIR    = ICODE             # directory for I-code files
ODIR    = OBJS              # directory for object file

# set flags for compile stages

CFLAGS  = -td=$(TEMP)
LFLAGS  = -olg -j

# list I-code files

IFILES  = $(IDIR)/main.i $(IDIR)/l_leg.i \
          $(IDIR)/r_leg.i $(IDIR)/l_arm.i \
          $(IDIR)/r_arm.i $(IDIR)/head.i \
          $(IDIR)/torso.i

# Special make rule to create optimized I-code files
.c.i :
    $(CC) $(CFLAGS) -eio=$(IDIR) $*.c

* compile into object code

$(ODIR)/$(OBJ) : $(IFILES)
    $(CC) $(IFILES) -fd=$(ODIR)/$(OBJ) $(CFLAGS)\
```

```
$(LFLAGS)
```

```
* include header file in all I-code files
```

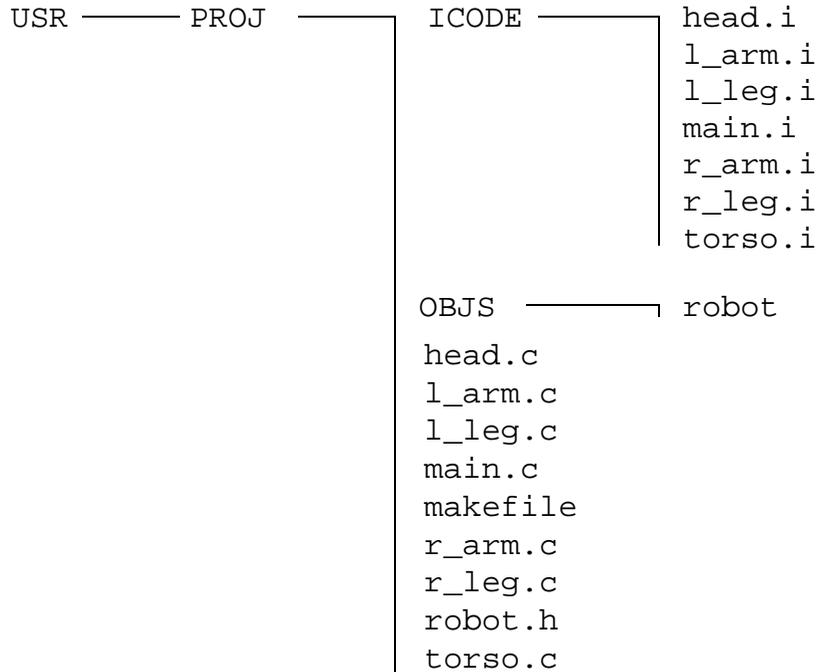
```
$(IFILES) : robot.h
```

Running this makefile produces the following command lines:

```
$ os9make
xccc -td=C:\TEMP -eio=ICODE main.c
xccc -td=C:\TEMP -eio=ICODE l_leg.c
xccc -td=C:\TEMP -eio=ICODE r_leg.c
xccc -td=C:\TEMP -eio=ICODE l_arm.c
xccc -td=C:\TEMP -eio=ICODE r_arm.c
xccc -td=C:\TEMP -eio=ICODE head.c
xccc -td=C:\TEMP -eio=ICODE torso.c
xccc ICODE/main.i ICODE/l_leg.i ICODE/r_leg.i ICODE/
    l_arm.i ICODE/r_arm.i ICODE/head.i ICODE/
    torso.i -fd=OBJS/robot -td=C:\TEMP -olg -j
```

The I-code linking makefile in the example on the previous page uses the directory structure shown in the following diagram.

Figure 4-1 I-Code Linking Makefile Example Directory Structure



Multiple Source File Program Object Code Linked with Makefile

An example makefile to use the object code link method might look like the following:

```

RDIR = RELS                                # Directory for ROF files
CFLAGS = -td=C:\TEMP                       # Use TEMP directory
CC = xcc

PROG = prog                                # Module to create
FILES = $(RDIR)/main.r $(RDIR)/misc.r     # All the .r files

# object code link and produce $(PROG)
$(PROG) : $(FILES)
    $(CC) $(CFLAGS) $(FILES) -f=$(PROG)
    
```

The following is a sample run of this makefile:

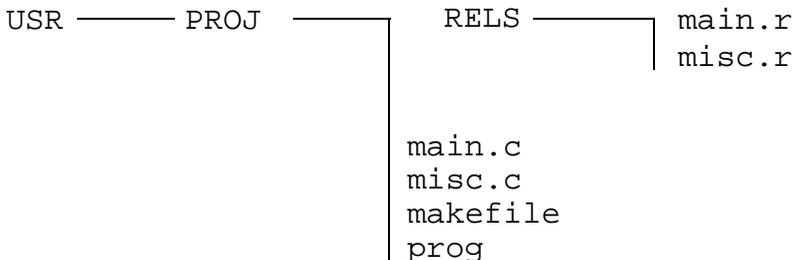
```

$ os9make
xcc -td=C:\TEMP main.c -eas=RELS
xcc -td=C:\TEMP misc.c -eas=RELS
xcc -td=C:\TEMP RELS/main.r RELS/misc.r -f=prog
    
```

Note that both source files were compiled into their ROF forms and then were object code linked and compiled into the module `prog`.

The object code linking makefile in the example uses the directory structure shown in the following diagram.

Figure 4-2 I-Code Linking Makefile Example Directory Structure



Multiple Source File Program I-Code Linked into Segments with Makefile

In this example (modified version of the previous I-code example) the makefile is used to I-code link parts of the application in subdirectories and then I-code link the mainline and the parts into the entire application. This method of creating an application enables physical separation of the source files.

The makefile used for this example is:

```
#
# makefile for Robot Controller
#
-o

TEMP    = C:\TEMP
CC      = xcc
OBJ     = robot
IDIR    = ICODE
ODIR    = OBJS
CFLAGS = -td=$(TEMP)
LFLAGS = -olg -j

IFILES = $(IDIR)/main.i LIMBS/limbs.il BODY/body.il

# Special make rule to create optimized I-code files
.c.i :
    $(CC) $(CFLAGS) -eio=$(IDIR) $*.c

$(ODIR)/$(OBJ) : $(IFILES)
    $(CC) $(IFILES) -fd=$(ODIR)/$(OBJ) $(CFLAGS)\
    $(LFLAGS)

$(IDIR)/main.i : robot.h
# make subdirectory I-code files into LIMBS/limbs.il

LIMBS/limbs.il : ./LIMBS
    cd LIMBS ; os9make

# make subdirectory I-code files into BODY/body.il
BODY/body.il : ./BODY
```

```
cd BODY ; os9make

# makefile for body
#
-o
TEMP = C:\TEMP
CC = xcc
OBJ = body.il
IDIR = ICODE
ODIR = ../body
CFLAGS = -td=$(TEMP)

IFFILES = $(IDIR)/head.i $(IDIR)/torso.i
# Special make rule to create optimized I-code files
.c.i :
    $(CC) $(CFLAGS) -eio=$(IDIR) $*.c
$(ODIR)/$(OBJ) : $(IFFILES)
    $(CC) $(IFFILES) -eil -fd=$(OBJ)

#
# makefile for limbs
#
-o
TEMP = C:\TEMP
CC = xcc
OBJ - limbs.il
IDIR - ICODE
ODIR = ../limbs
CFLAGS = -td=$(TEMP)

IFFILES = $(IDIR)/r_arm.i $(IDIR)/l_arm.i $(IDIR)/r_leg.i
$(IDIR)/l_leg.i
#Special make rule to create optimized I-code files
.c.i :
    $(CC) $(CFLAGS) -eio=$(IDIR) $*.c
$(ODIR)/$(OBJ) : $(IFFILES)
    $(CC) $(IFFILES) -eil -fd=$(OBJ)
```

Running the makefile produces the following command lines:

```
$ os9make
xcc -td=C:\TEMP -eio=ICODE main.c
cd LIMBS ; os9make
xcc -td=C:\TEMP -eio=ICODE r_arm.c
xcc -td=C:\TEMP -eio=ICODE l_arm.c
xcc -td=C:\TEMP -eio=ICODE r_leg.c
xcc -td=C:\TEMP -eio=ICODE l_leg.c
xcc ICODE/r_arm.i ICODE/l_arm.i ICODE/r_leg.i ICODE/
    l_leg.i -eil -fd=limbs.il
cd BODY ; os9make
xcc -td=C:\TEMP -eio=ICODE head.c
xcc -td=C:\TEMP -eio=ICODE torso.c
xcc ICODE/head.i ICODE/torso.i -eil -fd=body.il
xcc ICODE/main.i LIMBS/limbs.il BODY/body.il -
    fd=OBJS/robot -td=C:\TEMP -olg -j
```


Multiple Source File Non-Program I-Code Linked with Makefile

In this example a driver module is created. A device driver uses a different root `psect` than a program module so the `-cs` option is used to specify the root `psect` to use. Each source file is compiled to its I-code format and those I-code files are linked to each other and the default I-code libraries.

The makefile used for this example follows:

```
#
# makefile for device driver
#
-o

TEMP    = C:\TEMP
CC      = xcc
OBJ     = driver
IDIR    = ICODE
ODIR    = OBJS

CFLAGS  = -td=$(TEMP)
LFLAGS  = -olg -j -cs=drvrpsect.r

IFILES  = $(IDIR)/init.i $(IDIR)/read.i \
          $(IDIR)/write.i $(IDIR)/gstat.i \
          $(IDIR)/sstat.i $(IDIR)/term.i

# Special make rule to create optimized I-code files

.c.i :
    $(CC) $(CFLAGS) -eio=$(IDIR) $*.c

$(ODIR)/$(OBJ) : $(IFILES)
    $(CC) $(IFILES) -fd=$(ODIR)/$(OBJ) $(CFLAGS) \
          $(LFLAGS)

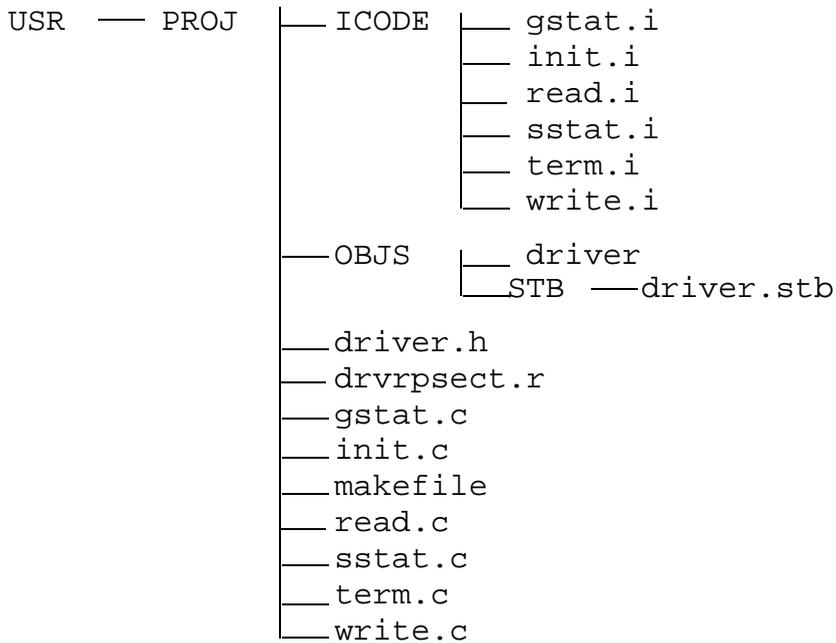
$(IFILES) : driver.h
```

Running this makefile produces the following command lines:

```
$ os9make
xcc -td=C:\TEMP -eio=ICODE init.c
xcc -td=C:\TEMP -eio=ICODE read.c
xcc -td=C:\TEMP -eio=ICODE write.c
xcc -td=C:\TEMP -eio=ICODE gstat.c
xcc -td=C:\TEMP -eio=ICODE sstat.c
xcc -td=C:\TEMP -eio=ICODE term.c
xcc ICODE/init.i ICODE/read.i ICODE/write.i ICODE/
    gstat.i ICODE/sstat.i
    ICODE/term.i -fd=OBJS/driver -td=C:\TEMP -olg -j -
    cs=drvrspect.r
```

The I-code linking makefile in the previous example uses the directory structure shown in the following figure.

Figure 4-4 Multiple Source File Non-Program I-Code Linked Example Directory Structure



Multiple Source File Object Library Creation with Makefile

Use the `make.olib` makefile to create the object code libraries:

```
#
# makefile for Robot Controller
#
-o

TEMP    = C:\TEMP
CC      = xcc
OBJ     = robotlib.l
RDIR    = RELS
ODIR    = LIB

CFLAGS = -td=$(TEMP)

RFILES = $(RDIR)/bend.r $(RDIR)/drop.r \
         $(RDIR)/grab.r $(RDIR)/move.r \
         $(RDIR)/rotate.r $(RDIR)/asm_code.r

$(ODIR)/$(OBJ) : $(RFILES)
    libgen $(RFILES) -co=$(ODIR)/$(OBJ)

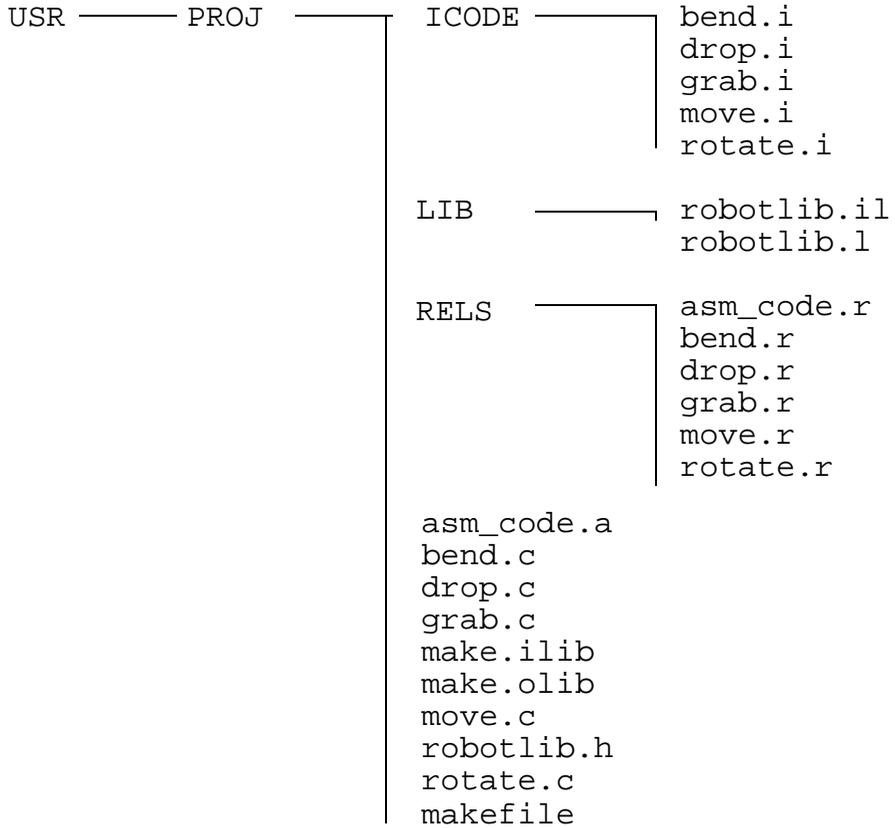
$(RFILES) : robotlib.h
```

Running the makefile produces the following command lines:

```
$ os9make -f=make.olib
xcc -td=C:\TEMP bend.c -eas=RELS
xcc -td=C:\TEMP drop.c -eas=RELS
xcc -td=C:\TEMP grab.c -eas=RELS
xcc -td=C:\TEMP move.c -eas=RELS
xcc -td=C:\TEMP rotate.c -eas=RELS
r68  asm_code.a -o=RELS/asm_code.r
libgen RELS/bend.r RELS/drop.r RELS/grab.r RELS/
      move.r RELS/rotate.r RELS/asm_code.r -
      co=LIB/robotlib.l
```

Both makefiles in the previous example assume the directory structure illustrated in the following table.

Figure 4-5 Multiple Source File Object Library Creation Example Directory Structure



Multiple Source File I-Code Library Creation with Makefile

The following example creates an I-code library from the same source files as were used in the previous example to create an I-code library.



Note

`asm_code.a` is not incorporated into the I-code library.

To use the back end data area layout facility, assembly language that defines global data or code symbols must reside in a source file by itself. This is because symbols defined in this manner are not resolved at I-code link time, and the back end requires resolution of all symbols before laying out the data area.

Use the `make.ilib` makefile to create I-code libraries:

```
#
# makefile for Robot Controller
#
-o

TEMP    = C:\TEMP
CC      = xcc
OBJ     = robotlib.il
IDIR    = ICODE
ODIR    = LIB

CFLAGS = -td=$(TEMP)

IFILES = $(IDIR)/bend.i $(IDIR)/drop.i \
          $(IDIR)/grab.i $(IDIR)/move.i \
          $(IDIR)/rotate.i

# Special make rule to create optimized I-code files
.c.i :
    $(CC) $(CFLAGS) -eio=$(IDIR) $*.c
```

```
$(ODIR)/$(OBJ) : $(IFILES)
    $(CC) -ilm $(IFILES) -eil -fd=$(ODIR)/$(OBJ) \
        $(CFLAGS)
```

```
$(IFILES) : robotlib.h
```

Running the makefile produces the following command lines:

```
$ os9make -f=make.ilib
xcc -td=C:\TEMP -eio=ICODE bend.c
xcc -td=C:\TEMP -eio=ICODE drop.c
xcc -td=C:\TEMP -eio=ICODE grab.c
xcc -td=C:\TEMP -eio=ICODE move.c
xcc -td=C:\TEMP -eio=ICODE rotate.c
xcc -ilm ICODE/bend.i ICODE/drop.i ICODE/grab.i ICODE/
    move.i ICODE/rotate.i
    -eil -fd=LIB/robotlib.il -td=C:\TEMP
```

**Figure 4-6 Multiple Source File I-code Library Creation Example
Directory Structure**

```
USR ——— PROJ ——— ICODE ——— bend.i  
                                     drop.i  
                                     grab.i  
                                     move.i  
                                     rotate.i  
  
LIB ——— robotlib.il  
  
RELS  
  
asm_code.a  
bend.c  
drop.c  
grab.c  
make.ilib  
make.olib  
move.c  
robotlib.h  
rotate.c  
makefile
```

Chapter 5: Compiler Phase Options

In addition to the options available for the executive, options may also be passed to the following compiler phases.

- **Front End**
- **I-Code Linker**
- **I-Code Optimizer**
- **Back End**
- **Assembly Optimizer**
- **Assembler**
- **Prelinker**
- **Object Code Linker**



For More Information

Chapter 11 contains additional information on optimizations.



Passing Options

Pass an option to a specific phase by prefacing the command line option (no spaces) with the appropriate phase code as identified in the following tables.

Table 5-1 ucc Executive Mode Compiler Phase Codes

Code	Phase
fe	Front end
il	I-code linker
io	I-code optimizer
be	Back end
ao	Assembly optimizer
as	Assembler
pl	Prelinker
ol	Object code linker



Note

When using an option such as `-E` in the front end, the proper syntax is:
`xcc -fe[=]E <files> <opts>`.

The hyphen on the “`-E`” is not included; similarly, if two hyphens are in front of an option, one hyphen is disregarded.

Table 5-2 c89 Executive Mode Compiler Phase Codes

Code	Phase
<code>Wp, W0</code>	Preprocessor/compiler
<code>Wi</code>	I-code linker
<code>W2</code>	I-code optimizer
<code>Wb</code>	I-code to assembly translator
<code>Wo</code>	Assembly optimizer
<code>Wa</code>	Assembler
<code>Wq</code>	Prelinker
<code>Wl</code>	Object code linker

Options and optimizations available for each phase are listed in this chapter, grouped by phase. The associated phase code is not prepended to command line options in the listings.



Note

Help for a phase is displayed by typing `xcc -<phase>?`.



Note

Every effort was made in the executive to eliminate the need to run phases “by hand”. Avoid including direct references to the phases in makefiles as the phase names and semantics are volatile. The executive, by its nature, remains more stable.

Front End



For More Information

For information on optimizations, see [Chapter 11](#).

The front end preprocesses and translates a C/C++ source file into I-code.

Command line options may be specified using single character option codes for example, `-o`, or keyword options for example, `--output`. A single character option specification consists of a hyphen followed by one or more option characters for example, `Ab`. If an option requires an argument, the argument may immediately follow the option letter, or may be separated from the option letter by white space. A keyword option specification consists of two hyphens followed by the option keyword for example, `--strict`. If an option requires an argument, the argument may be separated from the keyword by white space, or the keyword may be immediately followed by `=option`. When the second form is used there may not be any white space on either side of the equals sign.

The following options may be passed to the front end.

`--preprocess`

`-E`

Preprocess only, output goes to `stdout`

Do preprocessing only. Write preprocessed text to the preprocessing output file, with comments removed and with line control information.

`-Y<xfile>`

Generate extended cross-reference information in the file `>xfile<`.

`--no_line_commands`

`-P`

Preprocess only, output goes to stdout

Do preprocessing only. Write preprocessed text to the preprocessing output file, with comments removed and without line control information.

`--comments`

`-C`

Keep comments in the preprocessed output

This should be specified after either `--preprocess` or `--no_line_commands`; it does not of itself request preprocessing output.

`--dependencies`

`-M`

Generate list of dependency lines suitable for input to os9make

Do preprocessing only. Instead of the normal preprocessing output, generate on the preprocessing output file a list of dependency lines suitable for input to the `os9make` program.

`--trace_includes`

`-H`

Generate list of #include files, preprocess only

Do preprocessing only. Instead of the normal preprocessing output, generate on the preprocessing output file a list of the names of `#include` files.

<code>--define_macro <name[= value]></code>	
<code>-D[]<name[= value]></code>	Define <name> with an optional <value> Define macro <name> as value. If = value is omitted, define as 1. There are no macro names defined by default except for language-mandated macros like <code>__LINE__</code> .
<code>--undefine_macro <name></code>	
<code>-U[]<name></code>	Remove any definition of the macro <name> Remove any initial definition of the macro <name>. <code>--undefine_macro</code> options are processed after all <code>--define_macro</code> options in the command line have been processed.
<code>--include_directory <dir></code>	
<code>-I[]<dir></code>	Add directory to search list for #include files Add <dir> to the list of directories searched for <code>#include</code> files.
<code>--mt_enabled</code>	Compiling thread-enabled and thread-safe code Mark the resulting output file as thread-using and thread-safe.
<code>--mt_safe</code>	Compiling (non-thread-using) inherently thread-safe code Mark the resulting output file as inherently and thread-safe.
<code>--output <path></code>	Specifies the output file of the compilation For example, the preprocessing or intermediate language output file.
<code>--version</code>	Display edition number of the front end

`--no_code_gen`

Do syntax-checking only

For example, do not run the back end.

`--no_warnings`

`-w`

Suppress warnings

Errors are still issued.

`--remarks`

`-r`

Issue remarks

Issue remarks that are diagnostic messages milder than warnings.

`--error_limit <number>` **Set <number> of errors before aborting compilation**

Set the error limit to <number>. The front end abandons compilation after this number of errors (remarks and warnings are not counted toward the limit). By default, the limit is 100.

`--diag_suppress <tag,tag,...>`

`--diag_remark <tag,tag,...>`

`--diag_warning <tag,tag,...>`

`--diag_error <tag,tag,...>`

Override the error severity of specified diagnostic

Override the normal error severity of the specified diagnostic messages. The message(s) may be specified using a mnemonic error tag or using an error number. The error tag names and error numbers are listed in [Appendix A](#).

`--display_error_number` **Display the error number in diagnostic messages**

Display the error message number in any diagnostic messages that are generated. The option enables you to set the error number to be used when overriding the severity of a diagnostic message.

<code>--no_use_before_set_warnings</code>	
<code>-j</code>	Suppress warnings on local automatics that are used before their values are set Suppress warnings on local automatic variables that are used before their values are set. The front end algorithm for detecting such uses is conservative and is likely to miss some cases that an optimizer with sophisticated flow analysis could detect.
<code>--c++</code>	Enable compilation of C++ This is the default.
<code>--c</code>	Enable compilation of C rather than C++
<code>--old_c</code>	
<code>-K</code>	Enable K&R/pcc mode K&R/pcc mode approximates the behavior of the standard UNIX pcc. ANSI C features that do not conflict with K&R/pcc features are still supported in this mode.
<code>--strict</code> or <code>--strict_warnings</code>	
<code>-A</code> or <code>-a</code>	Enable strict ANSI mode, issue warnings for violations Strict ANSI mode provides diagnostic messages when non-ANSI features are used, and disables features that conflict with ANSI C or C++. This is compatible with both C and C++ mode (although ANSI conformance with C++ does not yet mean anything). It is not compatible with pcc mode. ANSI violations can be issued as either warnings or errors depending on which command line option is used. The <code>-A</code> and <code>--strict</code>

`--anachronisms`

`--cfront_2.1`

`-b`

`--cfront_3.0`

`--signed_chars`

`-s`

`--unsigned_chars`

options issue errors and the `-a` and `--strict_warnings` options issue warnings. The error threshold is set so that the requested diagnostics is listed.

Enable anachronisms in C++ mode

This option is valid only in C++ mode.

Enable C++ compatibility with cfront V2.1

Enable compilation of C++ with compatibility with `cfront` Version 2.1. This causes the compiler to accept language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (`cfront`) release 2.1. This option also enables acceptance of anachronisms.

Enable C++ compatibility with cfront V3.0

Enable Compilation of C++ with Compatibility with `cfront` Version 3.0. This causes the compiler to accept language constructs that, while not part of the C++ language definition, are accepted by the AT&T C++ Language System (`cfront`) release 3.0. This option also enables acceptance of anachronisms.

Make plain char signed

Default.

Make plain char unsigned

Support for unsigned characters is limited to only this feature. All library and

```
--suppress_vtbl
```

```
-V
```

`cs1` functions assume characters are signed by nature. These factors should be considered before using this option.

Suppress definition of virtual function tables

Suppress definition of virtual function tables in cases where the heuristic used by the front end to decide on definition of virtual function tables provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The `--suppress_vtbl` option suppresses the definition of the virtual function tables for such classes, and the `--force_vtbl` option forces the definition of the virtual function table for such classes.

`--force_vtbl` differs from the default behavior in that it does not force the definition to be local. This option is valid only in C++ mode.

`--force_vtbl`

Force definition of virtual function tables

Force definition of virtual function tables in cases where the heuristic used by the front end to decide on definition of virtual function tables provides no guidance. See `--suppress_vtbl`. This option is valid only in C++ mode.

`--instantiate <mode>`

Control instantiation of external template entities

External template entities, such as noninline and nonstatic, are external template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition.

This option is valid only in C++ mode.



For More Information

Information on template instantiation and exception disabling is documented in [Chapter 12: Language Features](#).

Table 5-3 Mode Descriptions

<code><mode></code>	Description
none	Instantiate no template entities. This is the default used.
used	Instantiate only the template entities that are used in this compilation.

Table 5-3 Mode Descriptions (continued)

<mode>	Description
<code>all</code>	Instantiate all template entities whether or not they are used.
<code>local</code>	Instantiate only the template entities that are used in this compilation, and force those entities to be local to this compilation.
<code>--no_implicit_include</code>	Disable implicit inclusion of source files as a method of finding definitions of template entities to be instantiated
<code>--no_auto_instantiation</code>	
<code>-T</code>	Disable automatic instantiation of templates
<code>--no_exceptions</code>	Disable support for exception handling
<code>--no_rtti</code>	Disable support for RTTI (runtime type information) features This option is valid only in C++ mode.
<code>--no_array_new_and_delete</code>	Disable support for array new and delete This option is valid only in C++ mode.
<code>--no_explicit</code>	Disable support for the explicit specifier on constructor declarations This option is valid only in C++ mode.
<code>--no_namespaces</code>	Disable support for namespaces This option is valid only in C++ mode.
<code>--no_using_std</code>	Disable implicit use of the std namespace when standard header files are included This option is valid only in C++ mode.

`--old_for_init`

Control the scope of a declaration in a for-init-statement

The old (`cfront`-compatible) scoping rules mean the declaration is in the scope where the `for` statement belongs. The new (standard-conforming) rules wrap the entire `for` statement in its own implicitly generated scope. This option is valid only in C++ mode.

`--no_old_specializations`

Disable acceptance of old-style template specializations

(Example: specializations that do not use the template `<>` syntax). This option is valid only in C++ mode.

```
--extern_inline  
--no_extern_inline
```

Disable support for inline functions with external linkage in C++

When `inline` functions are allowed to have external linkage, as required by the standard, then `extern` and `inline` are compatible specifiers on a nonmember function declaration; the default linkage when `inline` appears alone is external and an `inline` member function takes on the linkage of its class, which is usually external. In other words `inline` means `extern inline` on nonmember functions. However, when `inline` functions have only internal linkage, as specified in the ARM, then `extern` and `inline` are incompatible. The default linkage when `inline` appears alone is internal and `inline` member functions have internal linkage no matter what the linkage of their class. In other words `inline` means `static inline` on nonmember functions.

```
--dollar  
-$  
--error_output <efile>
```

Accept dollar signs in identifiers

Redirect what would have gone to `stderr` to file `efile`

- `--no_wchar_t_keyword` **Disable recognition of `wchar_t` as a keyword**
This option is valid only in C++ mode. The front end can be configured to define a preprocessing variable when `wchar_t` is recognized as a keyword. This preprocessing variable may then be used by the standard header files to determine whether a typedef should be supplied to define `wchar_t`.
- `--no_bool` **Disable recognition of `bool`**
This option is valid only in C++ mode. The front end can be configured to define a preprocessing variable when `bool` is recognized. This preprocessing variable may then be used by header files to determine whether a typedef should be supplied to define `bool`.
- `--no_typename` **Disable recognition of `typename`**
This option is valid only in C++ mode.
- `--no_implicit_typename` **Disable implicit determination, from context, of whether a template parameter dependent name is a type or nontype**
This option is valid only in C++ mode.
- `--old_style_preprocessing` **Forces `pcc` style preprocessing when compiling in ANSI C or C++ mode**
This may be used when compiling an ANSI C or C++ program on a system in which the system header files require `pcc` style preprocessing.

```
--xref <xfile>
```

```
-X <xfile>
```

Generate cross-reference information in the file <xfile>

For each reference to an identifier in the source program, a line of the form

```
<symbol-id> <name> X < file-name> <line-number>  
<column-number>
```

is written, where x is:

- D Definition
- d Declaration (a declaration that is not a definition)
- M Modification
- A Address taken
- U Used
- C Changed (meaning “used and modified in a single operation,” such as an increment)
- R Any other kind of reference
- E An error in which the kind of reference is indeterminate.

`symbol-id` is a unique decimal number for the symbol. The fields of the above line are separated by tab characters.

```
--xref2 <xfile>
```

```
-Y <xfile>
```

Generate extended cross-reference information in the file <xfile>

```
--list <file>
```

`-L <file>`

Generate raw listing information in the file `<file>`

This information is likely to be used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the front end. Each line of the listing file begins with a key character that identifies the type of line, as follows:

- N** A normal line of source. The rest of the line is the text of the line.
- X** The expanded form of a normal line of source. The rest of the line is the text of the line. This line appears following the **N** line, and only if the line contains non-trivial modifications. Comments are considered trivial modifications; macro expansions, line splices, and trigraphs are considered non-trivial modifications.
- S** A line of source skipped by an `#if` or similar statement. The rest of the line is text. Note that the `#else`, `#elif`, or `#endif` that ends a skip is marked with an **N**.

- L An indication of a change in source position. The line has a format similar to the # line-identifying directive output by `cpp`: such as:

L line-number "file-name" key

where key is 1 for entry into an include file, 2 for exit from an include file, and omitted otherwise. The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for #line directives (key is omitted). L lines indicate the source position of the following source line in the raw listing file.

R, W, E,C

An indication of a diagnostic (R for remark, W for warning, E for error, and C for catastrophic error). The line has the form:

```
<S> "file-name"  
<line-number><column-number><message-text>
```

Where S is R, W, E, or C, as explained above. Errors at the end of file indicate the last line of the primary source file and a column number of zero. Command-line errors are catastrophic errors with an empty file name (" ") and a line and column number of zero. Internal errors are catastrophic errors with position information as usual, and message-text beginning with (internal error). When a diagnostic displays a list such as, all the contending routines when there is ambiguity on an overloaded call, the initial diagnostic line

is followed by one or more lines with the same overall format (code letter, file name, line number, column number, and message text), but in which the code letter is the lower case version of the code letter in the initial line. The source position in such lines is the same as that in the corresponding initial line.

`--z [=<path>]`

Read file names and options from `stdin` or `<path>`

`--instantiation_dir <path>`

Directory where `.ii` files should go

`--target<n>`

Generate code for target processor/type number `<n>`



For More Information

Refer to [Chapter 2: Compiling](#) for more information on using `const` qualified pointers.

`--semantic_constraints_const_pointer`

Add semantic constraints to allow `const` qualifiers to appear in code area

Enable the addition of constraints on `const` qualified pointers allowing pointers to be stored in the code area. `Const` qualified pointers are treated differently than normal pointers, therefore their values cannot be assigned to or interpreted as normal pointers.

`--add_debug_info`

Include symbolic source level debugging information in I-code file

This information is eventually included in the symbol files generated by the object code link phase.

`--insert_source`

Include preprocessed source code as comments in I-code file

Include preprocessed source code as comments in the I-code file to enable output as comments in the assembly code by the back end.

`--Icode_buffer=<num>`

Set upper limit of I-code cache

Set the upper limit (in kilobytes) of the I-code cache. Generally, the larger this number, the faster the file compiles. The default cache size is 256K.

`--translate_names_1`

Translate names

Translates the names of certain library functions (mainly the `printf()` family of functions) for use with the Microware K & R C compiler libraries and `.r` files.

`--translate_names_2`

Translate names

Translate names of certain library functions to maintain compatibility with Microware K & R C compiler.

`--Extended_ANSI`

Extended ANSI source mode

`--class_browse`

create class browser data file

I-Code Linker

The I-code linker merges multiple I-code files into one I-code file. It also creates or links with an I-code library. In this module, an entire multi-source file application may be linked into one I-code file.

Any of the following options may be passed to the I-code linker.

`-b=<num>`

Set upper limit of I-code cache

Set the upper limit (in kilobytes) of the I-code cache. Generally, the larger this number, the faster the files link. The default cache size is 128K.

`-c`

Copy the I-code file.

`-l[=<file>`

Specify I-code library to link with linked I-code file

Specify an I-code library (created with the `-m` option) for use. The libraries are searched in the order specified on the command line.

`-m`

Generate I-code library from given I-code files

Create a file that can be used with `-l`.

`-mts`

Mark (non-thread-using) output library as inherently thread-safe

This option should be used when creating an i-code library that is inherently thread-safe and can be linked against either thread-using or non-thread-using libraries and intermediate files.

-mt [ewq]

Error level for thread-incompatibilities

e Emit errors

w Emit warnings and continue

q Quietly link mixed code

When creating an i-code library, no thread- incompatibilities are allowed.

-o [=]<file>

Write output to <file>

-v

Verbose warning for type mismatches

-z [=]<file>]

Read additional arguments (one per line) from standard input or <file>

The arguments are processed as if they were specified in place of the -z argument where the first line is the left-most argument and the last line is the right-most argument.

I-Code Optimizer

The I-code optimizer performs the language-independent and machine-independent optimizations. The I-code optimizer performs the following optimizations.

- Constant propagation
- Constant folding
- Variable lifetimes
- Common subexpression elimination
- Pointer tracking with common subexpression elimination
- Useless code, copy, and pointer eliminations
- Assignment translation
- Code motion and combining
- Initial loop condition testing
- Invariant hoisting
- Strength reduction
- Loop unrolling
- Constant sharing
- Function inlining

The following options may be passed to the I-code optimizer.

`-b=<num> [k]`

Set upper limit of I-code cache

Set the upper limit (in kilobytes) of the I-code cache. Generally, the larger this number, the faster files optimize. The default cache size is 64K.

- `-c`

Suppress common subexpression elimination optimization
Suppress the common subexpression elimination (CSE) optimization. `-c` allows several possible sub-options to control its behavior.
- `-cd`

Suppress CSE of one double identifier or constant
It may be advantageous to move a double identifier or constant into a register. However, on some processors this can be more expensive than re-accessing the double in its original storage.
- `-cg`

Suppress CSEs of expressions containing global identifiers
Although the I-code optimizer is conservative in this respect, some programs may disallow creation of temporaries to hold the result of a common subexpression containing global identifiers (example, a program with a signal handler that modifies a non-volatile qualified global).
- `-cl`

Suppress CSEs containing dangerous local variables
Dangerous local variables are local variables with an address stored in another location or passed to another function that may cause asynchronous modification. This type of local variable is treated much like a global variable.

-ct

Suppress pointer-tracking pass before CSE

Normally, the optimizer makes a pass over a function before performing CSE to determine which objects are modified by pointer indirection. Pointer tracking allows greater compiler accuracy concerning what can be kept in a register.

-e

Suppress all processor specific optimizations

Suppress all processor specific optimizations. -e allows several possible sub-options to control its behavior.

-ec

Suppress creation of CSEs containing worthwhile constants

For some processor architectures it may be advantageous to consider certain eligible constants for common subexpression elimination; the recomputation of these values may be expensive.

-ef

Suppress creation of CSEs containing function addresses

For some processor architectures it may be advantageous to consider the computation of the absolute address of a function to be eligible for common subexpression elimination; the recomputation of this value may be expensive.

-eg

Suppress creation of CSEs containing global variable addresses
For some processor architectures it may be advantageous to consider the computation of the address of a global variable to be eligible for common subexpression elimination; the recomputation of this value may be expensive.

-em

Disable memory storage in registers
For processors with 32 or more general purpose registers, the contents of memory loaded via simple addressing modes (base register or base register + offset) will be stored in registers. Previously, memory loaded this way was not stored in a register. The option `-em` (`-e` option with sub-option `m`) has been added to disable this new feature.

-h

Suppress code motion and combination optimizations

-i [=] <num>

Specify maximum number of useless-code elimination passes
Each useless-code elimination pass can affect the code such that more useless code can be eliminated. `<num>` specifies the maximum number of passes.

- If `<num>` is zero, code elimination passes are disabled.
- If `<num>` is greater than fifteen, all useless code is eliminated.

`-k [=]<space> : <time>`

Specify importance of space and time for file being optimized

If time is more important than space, the optimizer allows the code size to increase if it saves time. `-k` primarily affects function inlining and loop unrolling.

`<space>` and `<time>` are strings of decimal digits or empty strings. If either field is left blank, the default value of one is used. If either value is zero, the other value is the maximum value.

`R-l`

Suppress variable lifetime computations

Lifetime information is computed for all local variables by default. This information helps the backend in performing register allocation.

`m`

Allow inliner to discard single-call functions

A single-call function is one that is called only once by the other functions in the I-code file. If any function in the I-code file is called by an external function, do not use this option.

`-n`

Suppress global function inlining

Function inlining is enabled by either specifying that time is more important than space or using `-m` to enable discarding of single-call functions. This option overrides automatic function inlining.

`-o`

Optimize I-code files

Suppress all loop optimizations.

`-oi`

Suppress invariant relocation loop optimization

<code>-os</code>	Suppress strength reduction loop optimization
<code>-ou</code>	Suppress loop unrolling optimization
<code>-p</code>	Suppress constant propagation and folding
<code>-r</code>	Enable function return value checking The optimizer checks the return statements on all functions. If the function is declared to return a value and it contains a return with no expression, a warning is printed.
<code>-t</code>	Suppress translation of assignment statements
<code>-u</code>	Enable uninitialized variable warnings As a by-product of variable lifetime computation, the optimizer can determine if a variable is used before it is given a value. If this is the case, a warning is generated.
<code>-z[[=]<file>]</code>	Read command line arguments (one per line) from file If <code><file></code> is not specified, the command line arguments are read from standard input.

Back End

The back end translates an I-code file into assembly language and performs machine-dependent optimizations. The back end performs the following functions.

- Lays out the data area
- Selects code to generate based on time and space considerations
- Assigns registers according to the greatest need
- Generates code

The back end performs register coloring and coalescing optimization.



For More Information

Refer to the appropriate *Ultra C/C++ Processor Guide* for additional back end option descriptions.

The following non processor-specific options may be passed to the back end.

`-b[=]<num> [k]`

Set upper limit of I-code cache

Set the upper limit (in kilobytes) of the I-code cache. Generally, the larger this number, the faster the files are translated. The default cache size is 128K.

`-g`

Emit symbolic source level debugging information

The object code linker eventually writes the information to the debugging support files.

`-l=<pathlist>`

Search <pathlist> as library

`-m [=] <num> [k]`

Non-remote memory available

Identifies the amount of memory remaining before compiling the source code. This value is used when laying out the data area for the program. This memory can then be accessed with short code.

`-o [=] <pathlist>`

Specify output file for assembly code

`-s [=] <num>`

Specify space factor

Determines the back end code selecting algorithm. If space is greater than time, code that assembles into smaller, possibly slower, object code is selected when more than one option is available.

`-t [=] <num>`

Specify time factor

Determines the back end code selecting algorithm. If time is greater than space, code that assembles into faster, possibly larger, object code is selected when more than one option is available.

`-z [= <pathlist>]`

Read file names and options (one per line) from standard input or <pathlist>

Assembly Optimizer

The assembly optimizer performs machine-dependent optimizations. The following options may be passed to the assembly optimizer.

- | | |
|----------------------|--|
| -h | Suppress processor-specific pattern optimizations
-h is rarely required. |
| -i [=]<num> | Set size of instruction peephole
Sets the size of the instruction peephole to <num> instructions. The peephole is the number of instructions that the assembly optimizer examines as it sequentially scans the file. |
| -k [=]<space>:<time> | Specify importance of space and time related issues
This affects some optimizations that sacrifice time to save space. |
| -o [=]<file> | Write optimized output to <file>
If -o is not used on a command line, standard output is assumed. |
| -p | Do not perform processor specific optimizations |



For More Information

Refer to the *Ultra C/C++ Processor Guide* for assembly optimizer processor numbers and additional options.

- | | |
|-------------|---|
| -t [=]<num> | Specify target processor family |
| -v | Assume all memory reads are volatile |

```
-z[ [=]<file> ]
```

Read additional arguments (one per line) from standard input or <file>
The arguments are processed as if they were specified in place of the `-z` argument where the first line is the left-most argument and the last line is the right-most argument.

Assembler



For More Information

Refer to **Chapter 11** for information on span-dependent optimizations. Refer to the assembler specific chapters of this manual, **Chapter 6** and **Chapter 7** and the *Ultra C/C++ Processor Guide* for additional information about the assembler.

The assembler translates the assembly file into an ROF and performs span-dependent optimizations. The options identified following may be passed to the assembler.

-a [=]<sym> [= <val>]

Allow symbol to be defined before assembly begins

The symbol is defined as if it was specified as the label on a set directive. If a value is given, the label is set to that value. Otherwise, the default value of 1 is assumed. -a is most useful with the `ifdef` or `ifndef` directives.

-c

List Conditional Assembly Lines in Assembler Listing

By default, this option is off.

-d<num>

Set number of lines per page

Sets the number of lines per page for listing to <num>. The default is 66.

NOTE: This option is ineffective without an accompanying -l option.

-e

Suppress printing of errors

By default, this option is off.

- f** **Use form feed for page Eject, instead of line feeds**
Use form feed for top of form. By default, this option is off.
This option is ineffective without an accompanying `-l` option.
- g** **List all generated code bytes**
By default, this option is off.
- k** **Force the assembler to keep the output file, even if there were errors generating it**
This only applies if `-o` is used to create an output file.
- l** **Write formatted assembler listing to standard output**
If not used, only error messages are printed. By default, `-l` is off.
- mt** **Compiling thread-enabled and thread-safe code**
Mark the resulting output file as thread-using and thread-safe.
- mts** **Compiling (non-thread-using) inherently thread-safe code**
Mark the resulting output file as inherently and thread-safe.
- n** **Omit line numbers from assembler listing**
This allows more room for comments. This option is ineffective without an accompanying `-l` option.
- o=<path>** **Write relocatable output to specified file**
The specified file must be a mass storage file. By default, this option is off.

-p<n>

Align orgs to boundary

Align all section bases to <n> boundary. <n> may be 2, 4, 8, or 16 bytes.

-q

Quiet mode

Suppress warnings and nonfatal messages. By default, this option is off.

-s

Print symbol table at end of assembler listing

By default, this option is off.

-u[=]<dir>

Add directory to search list for use statement

-v

Display assembler version and edition number on standard error path

By default, this option is off.

-x

Print macro expansion in assembler listing

By default, this option is off.

NOTE: This option is ineffective without an accompanying -l option.

-z[=]<file>

Use <file> for input arguments (one per line)

The arguments are processed as if they were specified in place of the -z argument. The first line is the left-most argument and the last line is the right-most argument.

Prelinker

The following options can be passed to the prelinker:

- m** **Do not demangle identifier names that are displayed**
- n** **Update the instantiation list but do not recompile the files**
- D** **Do not assign instantiations to non-local object or I-code files**
Instantiations may be assigned only to object files in the current directory.
- N <file>** **If a file from a non-local directory needs to be recompiled, perform the compilation in the current directory**
An updated list of object files and library names is written to the file specified by <file> enabling the compiler to identify alternate versions of object files that should be used.



For More Information

Refer to the [Automatic Instantiation](#) section in [Chapter 8](#).

- r** **Do not stop after a certain number of iterations**
- q** **Turns off verbose mode**

`-s <nnn>`

Specifies whether the prelinker should check for entities that are referenced as both explicit specializations and generated instantiations

If `nnn` is zero, the check is disabled (default); otherwise, the check is enabled.

`-S`

Suppress instantiation flags in the object files

This causes the prelinker to recompile all of the local object files passing the flag-suppression option to the front end.

`-filink`

Prelink I-code files and libraries

Object Code Linker



For More Information

Refer to [Chapter 6](#), [Chapter 9](#), and the *Ultra C/C++ Processor Guide* for additional information about the object code linker.

The object code linker links ROFs with libraries to produce an OS-9 module. The following options may be passed to the object code linker.

`-b=<n>`

Align code and data segments

Align code and data segments to `<n>`. `<n>` may be 2, 4, 8, or 16 bytes.

`-c`

Make linker perform in ANSI conformant manner

If this option is specified, names of labels made by the linker begin with an underscore (`_`).

`-e=<num>`

Set module edition number

`<num>` is used for the edition number in the final output module. The mainline edition number of `psect` is used if `-e` is not given.

`-f=<perms>`

Set additional file permissions

Valid `<perms>` are:

<code>pr</code>	= Public read
<code>pw</code>	= Public write
<code>pe</code>	= Public execute
<code>gr</code>	= Group read
<code>gw</code>	= Group write
<code>ge</code>	= Group execute
<code>or</code>	= Owner read
<code>ow</code>	= Owner write
<code>oe</code>	= Owner execute

`-g`

Output symbol modules for use by user and/or source debugger

If the `.r` files were created with the `-g` option, two symbol files are created:

- one file name with `.stb` appended
- one file name with `.dbg` appended

If not compiled with `-g`, only the `.stb` file is created. If a directory named `STB` is present in the current execution directory, the symbol files are placed there. Otherwise, they are placed in the current execution directory.

`-gu=<group>.<user>`

Set module owner

Set the owner of the final module to user `# <user>` in group `# <group>`. Both `<group>` and `<user>` must be specified in order to transfer ownership.

`-i`

Generate system module with initialized data

Extend the definition of OS-9 for 68K system modules to allow initialized data.

`-l=<path>`

Use <path> as library file

A library file consists of one or more merged assembly ROFs. Each `psect` in the file is checked to see if it resolves any unresolved references. If so, the module is included in the final output module, otherwise, it is skipped.

Mainline `psects` are not allowed in a library file. This option may be repeated up to 32 times in one command line to specify multiple library files. Library files are searched in the order given on the command line. The compiler standard definition files are `sys.l` for assembly language or `clib.l`.

`-M=<mem>[k]`

Add <mem>K to stack memory allocation

`-m[=<path>]`

Print linkage map

Print the linkage map indicating the base addresses of the `psects` in the final object module. The output is directed to `stdout`, unless `<path>` is specified.

`-mt[ewq]`

Error level for thread-incompatibilities

`e` Emit errors

`w` Emit warnings and continue

`q` Quietly link mixed code

`-n=<name>`

Use <name> as module name

`-o=<path>`

Write linker object output to specified file

Write linker object (memory module) output to the specified file, relative to the execution directory. The last element in `<path>` is the module name unless overridden by the `-n` option.

`-O=<path>`

Write linker object output to specified file

Write linker object (memory module) output to the specified file, relative to the data directory. The last element in `<path>` is the module name unless overridden by the `-n` option.

`-p=<num>`

Set permission word

Set the permission word in the module header to `<num>`. `<num>` must be hexadecimal.

`-r`

Output raw binary file for non-OS-9 target system

The output is not in memory module format.

-r=<num>

Output Raw Binary File

Output a raw binary file for non-OS-9 target systems with an object code base address at absolute address <num>. <num> must be a hexadecimal address.

This option may also be used to make OS-9 boot ROM.

-R=<n>

Set the module revision number

Set the module revision number to <n>, where <n> is less than 255.

-s [=<path>]

Print list of relative addresses assigned to symbols

Print a list of relative addresses assigned to symbols in the final object module. The symbols are listed in numeric order. -s is generally used with the -m option. Output is directed to stdout, unless <path> is specified.

-S

Set sticky bit in module header

This causes the module to remain in the module directory, even if the link count becomes zero.



For More Information

Refer to the *Ultra C/C++ Processor Guide* for target code/module type descriptions.

-t=<target>

Specify target module type

-v

Verbose mode

Print warning messages if needed.

`-w`

Display symbols in alphabetic order

When used with the `-s` option, it displays symbols in alphabetic instead of numeric order.

`-x=<n>`

Align execution offset

Align execution offset to `<n>` boundary. `<n>` may be 2, 4, 8, or 16 bytes.

`-z`

Read module names from standard input (one per line)

`-z=<file>`

Read module names from `<file>` (one per line)

The arguments are processed as if they were specified in place of the `-z` argument. The first line is the left-most argument and the last line is the right-most argument.

Chapter 6: Assembler and Object Code Linker Overview

This chapter contains information on the following topics:

- **Assembler**
- **Linker**
- **The Assembly Language Program Development Process**
- **Relocatable Program Sections**
- **Program Section Declarations: psect and vsect**
- **ROF Edition Number 9 Format**
- **ROF Edition Number 15 Format**



Assembler

The assembler accepts an assembly language file for input and outputs an ROF. An ROF contains information such as the global data definitions, code entry points, external references, actual object code, and initialized data.



For More Information

Refer to the **Relocatable Object File Format** subsection in this chapter for more information about ROFs.

The assembler is processor specific. Use the assembler name that is appropriate for your processor as identified in **Table 6-1**.



For More Information

Refer to **Chapter 7**, for more information.

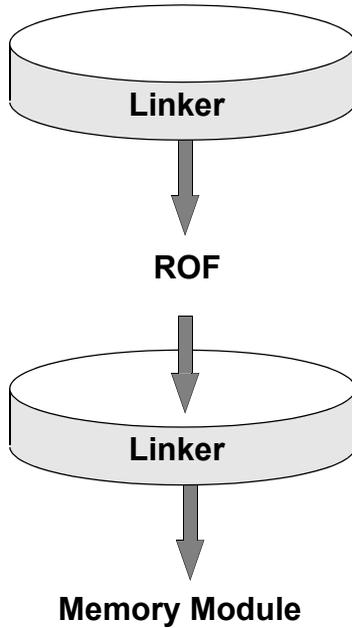
Table 6-1 Macro Assembler Names

Target Processor	Macro Assembler Name
68K	r68
80386/80486/Pentium	a386
PowerPC	appc
ARM	aarm
SH	ash

Linker

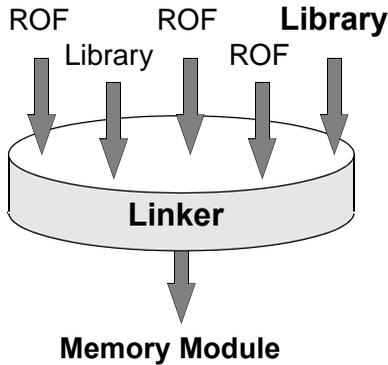
The linker takes the ROF produced by the assembler and produces an OS-9 memory module as shown in **Figure 6-1**.

Figure 6-1 Memory Module



The linker can also take multiple ROFs and libraries and produce a single memory module as shown in [Figure 6-2](#).

Figure 6-2 Multiple ROF Memory Module



The ROFs provide the linker with the information required to create each type of memory module. The linker allows code in one ROF to reference a symbol in another ROF. This involves adjusting the operands of the machine-language instructions referencing the symbol.

A library contains one or more ROFs merged into a single file. A library is created using one of two methods:

- Concatenating one or more ROFs using the `merge` utility into a single file
- Using the `libgen` utility

`libgen` creates libraries that contain index and cross reference information to speed the linking process. Once a library is created with `libgen`, `libgen` may be used to display the library information.



For More Information

Refer to [Chapter 10](#), and the *Utilities Reference* for description of utilities for the assembler and linker.

The linker is processor specific. Use the linker name, as identified in [Table 6-2](#), appropriate for your processor.



For More Information

Refer to [Chapter 9](#) for more information.

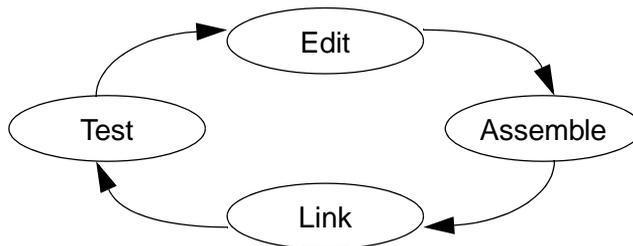
Table 6-2 Linker Names

Target Processor	Linker Name
68K	l68
80386/80486/Pentium	l386
All other OS-9 processors supported in the future	linker

The Assembly Language Program Development Process

The writing and testing of assembly language programs using the assembler and the linker involves a basic edit, assemble, link, and test cycle.

Figure 6-3 Assembly Language Program Development Process



The assembler and linker can simplify this process if programs are written in sections that may be assembled separately and later linked together to form the entire program. This way, if one program section requires change for any reason, only the revised section requires reassembly.

The following is a summary of the steps involved in the assembly language development process.

1. Create a source program file using a text editor.
2. Run the assembler to translate the source file(s) to ROF(s).
3. If necessary, use the text editor to correct errors reported by the assembler and repeat step 2.
4. Use the linker to combine all required relocatable modules. If the linker reports errors, correct them and repeat step 2.
5. Run and test the program. The debugger may be used to test programs.
6. If bugs are found in the program, use the text editor to correct the source file and repeat steps 2 through 5.

Relocatable Program Sections

A primary purpose of the assembler is to permit programs to be composed of different segments (source files) that may be assembled separately.

OS-9 processes use at least two separate areas of memory:

- the program object code in memory module format
- a data area used for the program's variables and the stack

The linker combines all of the source files into a single OS-9 for 68k or OS-9 memory module and a coordinated data storage area. By using global symbolic names, code in each source file can reference variables declared in other source files or may transfer program control to labels in other source files.

When the assembler source program for each source file is written, it must be divided into distinct sections for variable storage declarations sections (`vsects`) and for program instruction sections (`psects`). The output of the assembler is an ROF containing the object code output and information about the variable storage declarations for use by the linker.

Program Section Declarations: `psect` and `vsect`

Most program statements are included in sections called `psects` and `vsects`.

- A `psect` (program section) contains the program instructions and code area constants.
- A `vsect` (variable declaration section) contains the variable storage declarations for use by the linker.

`psects`

The `psect` directive begins the program section. An `ends` or `endsect` statement ends the program section. The `psect` directive initializes all assembler location counters and marks the start of the program source file.

Each source file may have only one `psect`. Global symbols (labels with a colon (`:`) suffix) in this section are accessible from all other program source files. Similarly, statements in this section can reference global symbols properly defined in other program source files. Statements in this section may also appear in linkage maps and symbolic debugger symbol lists.



For More Information

Chapter 7 contains more information about the `psect` directive.

There are two types of `psect`s:

- The Root `Psect`

The root `psect` is the `psect` that determines the type of module the linker creates. The file containing the root `psect` must be named first on the linker command line. The root `psect` is distinguished from other `psects` by a non-zero type/language field in the `psect` directive of the source file. All other `psects` processed in the linkage must have a zero type/language field.

The `psect` directive in the root `psect` provides the linker with the name of the `psect` and preliminary module header information. The type/language, attribute/revision, edition number, stack requirement, module entry point offset, and uninitialized trap handler entry point also appear in the `psect` directive. They set up the corresponding entries in the module header. Linker command line options may be used to change these values.

- The Subroutine `Psect`

A zero type/language field in the `psect` directive indicates a subroutine `psect`. These `psects` are usually subroutines that provide supporting code for the root `psect`. Linker library files are separately assembled `psects`, merged or generated (by `merge` or `libgen`, respectively) together into a single file. Except for the `psect` name and stack size reservation fields, all fields in the `psect` directive are zero.

vsects

A variable declaration section begins with a `vsect` directive and ends with an `ends` or `endsect` statement.

Global and local variable storage is declared inside one or more `vsects` within the `psect`. `vsects` are usually nested within a `psect`. `vsects` cannot be nested within themselves.



For More Information

Chapter 7 contains more information about the `vsect` directive.

Many processor architectures support addressing modes consisting of a base register plus an offset where there are options for the size of the offset. For example, the Motorola 68020 has one addressing mode that is a base register plus a 16-bit offset and another addressing mode that is base register plus a 32-bit offset. Instructions that use 16-bit offsets are shorter and sometimes faster than those that use 32-bit offsets. However, 16-bit offsets can only reach addresses that are within $\pm 32K$ of the base register. Under OS-9, where the data area is always accessed via the data area pointer, a Motorola OS-9 for 68K processor can access 64K of the data area by using the data area pointer and a 16-bit offset. If there is more than 64K of data, it must be addressed by using a longer addressing mode.

To aid in organizing the data area such that large arrays and structures do not consume the limited part of the data area that can be addressed by shorter addressing modes and to enable frequently used variables to reside in that area, there are two types of `vsects`: normal data `vsects` and remote data `vsects`.

- Normal Data `Vsects`

Normal data `vsects` allow allocation of variables for which a short addressing mode may access. However, the amount of normal data `vsects` is limited. For example, on Motorola 68K machines, the amount of normal data `vsects` is limited to 64K.

- Remote Data `Vsects`

This type of `vsect` pertains to remote data. In this section, remote data is somewhat processor specific. Some processors treat remote data essentially as non-remote data.

- The linker places remote `vsect` declarations after the end of all the normal `vsect` declarations. The size of the remote data area is limited only by the amount of physical memory on the system. However, a longer addressing mode must be used to access remote data `vsects`.

Variable declarations can be either initialized or uninitialized:

- **Initialized Variables**

This corresponds to the assembly language storage allocation mnemonic `dc`. The linker combines all initialized variable declarations from all program source files into a single initialized data memory area.

- **Uninitialized Variables**

This corresponds to the assembly language storage allocation mnemonic `ds`. All uninitialized data declarations are combined into a single uninitialized data memory area.

Certain types of statements can appear outside, usually before, the `psect`. These statements are generally `set` and `equ` (and possibly the `lo` and `do` statements). These declare symbolic constants and symbolic offsets. Labels on these statements are local to the assembly of the source file and are not functional during assembly of other program source files. Additionally, these statements cannot reference any global symbols.

Although technically a `vsect` can appear outside the `psect`, the usefulness of such a `vsect` is limited to defining the expected type of an external symbol as a data area symbol because actual storage would not be assigned to it by the linker.

Example Program Layout for a Motorola OS-9 for 68K Program

Table 6-3 Program Layout for Motorola OS-9 for 68K

Description	Example source nam Data
nam example	ttl sections and declarations
Local constant definitions	labconst equ 1 space equ \$20 mode set 1
Include local definitions file	use /h0/defs/oskdefs.d
Start of psect	psect nam,typ,rev,ed,stack,gblcode
Start of normal vsect	vsect
Global uninitialized data	gblunin: ds.l 1
Global initialized data	gblinit: dc.b "string",0
Local uninitialized data	locunin ds.l 1
Local initialized data	locinit dc.b "hello",0
End of vsect	ends

Table 6-3 Program Layout for Motorola OS-9 for 68K (continued)

Description	Example source nam Data
Start of remote vsect	vsect remote
Global remote uninitialized data	rgblunin: ds.l 1
Global remote initialized data	rgblinit: dc.b "string",0
Local remote uninitialized data	rlocunin ds.l 1
Local remote initialized data	rlocinit dc.b "hello",0
End of remote vsect	ends
Global code label	gblcode:
Global uninitialized data reference	move.l gblunin(a6),d0
Global initialized data reference	lea gblinit(a6),a0
Local code reference	bsr.s intcode
External code reference	bsr extcode

Table 6-3 Program Layout for Motorola OS-9 for 68K (continued)

Description	Example source nam Data
External data reference	<code>move.l d0,extdat(a6)</code>
Global uninitialized data reference	<code>move.l #rgblunin,d0</code> <code>move.l (a6,d0),d0</code>
Global initialized data reference	<code>move.l #rgblinit,d0</code> <code>lea (a6,d0),a0</code> <code>rts</code>
Local code label	<code>intcode rts</code>
End of psect	<code>ends</code>

Location Counters

The assembler maintains a set of address counters that track the relative memory addresses for the following types of code and data:

- Object code
- Initialized data
- Uninitialized data
- Remote initialized data
- Remote uninitialized data



Note

Location counter values are relative offsets and not the actual physical memory addresses. Actual memory locations are not known until the program has been linked and loaded into memory.

The `psect` statement resets the instruction and data location counters. As object code is generated, the instruction location counter is advanced accordingly. Outside of a `vsect` an asterisk (*) may be used in expressions to refer to the current relative value of this counter.

The `vsect` statement causes the assembler to use the variable (data) location counters and places information about subsequently declared variables in the appropriate ROF data description area. As variables are declared, the initialized data location counter and the uninitialized data location counter are advanced accordingly. Within a `vsect`, an asterisk (*) represents the value of the initialized data location. The uninitialized data location may not be accessed directly.

Counters may not be preset to a specific value. That is, there is no `org` statement for the data or instruction counters.

Relocatable Object File Format

The linker must process the assembler object code output to create executable code. The assembler writes the object code in a special ROF format to allow the linker to link separately assembled modules into a single executable module. The ROF contains information such as the global data definitions, code entry points, external references, actual object code, and initialized data.

It is unlikely that you will have to deal with the internals of an ROF. The `rdump` and `libgen` utilities may be used to extract this data from existing relocatable files and libraries.

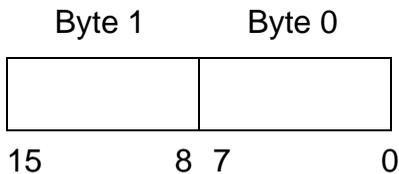


For More Information

Refer to **Chapter 10** for more information about utilities for the assembler and linker, including `rdump`.

In this section, bytes are numbered as shown in **Figure 6-4**.

Figure 6-4 Byte Numbering



Two different ROF formats are used by Microware assemblers: Edition Number 9 and Edition Number 15. See the ***Ultra C/C++ Processor Guide*** to determine the edition number format the assembler for your processor uses.

ROF Edition Number 9 Format



Note

This section refers to Editions 9.1 and 9.2.

ROF Edition Number 9 comprises these sections:

- **Header Section**
- **Object Code Section**
- **Initialized Data Section**
- **Remote Initialized Data Section**
- **Debug Information Section**
- **External Definition Section**
- **Local Reference Section**

Header Section

ROF Sync Bytes (4 bytes)

Sync bytes used by the linker to recognize an ROF.

Type/Language (2 bytes)

The type/language word from the `psect`. The linker uses the type and language to determine the desired module format. If this word is zero, the routine is assumed to be a subroutine type module. Only the root `psect` can have this word be non-zero.

Attribute/Revision (2 bytes)

Attribute/revision word to place in the module. It is only meaningful on a root `psect`.

Assembly Valid (2 bytes)

Word used to prevent the linker from linking erroneous modules. It is non-zero if assembly errors have occurred.

Series (2 bytes)

Informs the linker of the assembler version used. This prevents problems that could occur in mixing different versions of the linker and the assembler.

Date/Time Assembled (6 bytes)

Indicates the date and time of assembly.

Edition number (2 bytes)

A user-definable edition number to place in the output module for root `psects`. For non-root `psects`, this word is informational only.

Size of Static Storage (4 bytes)

Informs the linker of the amount of static data storage to reserve for the module. This value is determined from the total size of the `ds` directives in the `vsects`.

Size of Initialized Data (4 bytes)

Informs the linker of the amount of initialized data contained in the module. The size is determined by the total size of all `dc` directives in the `vsects`.

Size of the Object Code (4 bytes)

This is determined from the size of the assembled code.

Size of Stack Required (4 bytes)

Informs the linker of the amount of stack space the module requires. The value is obtained directly from the `psect` directive.

Offset to Entry Point (4 bytes)

Offset to the entry point in the object code. The offset is relative to the beginning of the module. The value is obtained directly from the `psect` directive.

Offset to Uninitialized Trap Entry Point (4 bytes)

Offset to the entry point in the object code that is called when a `tcall` is made without installing the appropriate trap handler. The offset is relative to the beginning of the module and is obtained directly from the `psect` directive.

Size of Remote Static Storage (4 bytes)

Informs the linker of the amount of remote static data storage to reserve for the module. This value is determined from the total size of the `ds` directives in the remote `vsects`.

Size of Remote Initialized Data (4 bytes)

Informs the linker of the amount of remote initialized data contained in the module. The size is determined by the total size of all `dc` directives in the remote `vsects`.

Size of the Debug Information (4 bytes)

Determined from the size of the assembled debugger code.

Code Information (2 bytes—9.2)

Information about the code in the file may be found here. For example, whether the code is thread-using or thread-safe.

Header Expansion (2 bytes—9.2)

Reserved.

Name of Module (variable length)

Null terminated ASCII string taken directly from the `psect` directive. The linker uses the name to identify the `psect` in case of an unresolved reference or other error.

External Definition Section

External Definition Count (2 bytes) (4 bytes—9.1)

Indicates the number of external definitions that follow. The external definitions are placed in the linker symbol table and can be referenced by any other module.

External Definitions (variable)

Each external definition has the following format:

- Name (2-n bytes)
- Type Definition (2 bytes)
- Symbol value (4 bytes)

Bits 0-2 determine the type:

Bit 20 Specifies data definition
 1 Specifies code definition

Code type flags:

Bits 0-1	00	Code label
	01	set label
	10	equ label

Data type flags:

Bits 0-1	00	Uninitialized non-remote
	01	Initialized non-remote
	10	Uninitialized remote
	11	Initialized remote

Bit 81 Common block definition

The meaning of the type definition values are:

0x0000	Uninitialized data
0x0001	Initialized data
0x0002	Uninitialized, remote data
0x0003	Initialized, remote data
0x0004	Code
0x0005	set
0x0006	equ
0x0100	Common data
0x0102	Remote common data

No other values are valid for an external type definition.

Object Code Section

Object Code for the Module (variable length)

The size of this section is found in the `Size of Object Code` defined in the header section.

Initialized Data Section

Initialized Data (variable length)

The size of this section is found in the `Size of Initialized Data` defined in the header section.



Note

The object code section and/or the initialized data section may be omitted. If both are missing and the static data count is zero, the module can only contain absolute (`equ`) symbols. In this case, the linker extracts only those symbols that resolve an external reference. The `sys.l` library module is an example. It contains only `equ` symbol definitions.

Remote Initialized Data Section

Initialized Remote Data (variable length)

The size of this section is found in the `Size of Remote Static Storage` in the header section.

Debug Information Section

Debug Information (variable length)

The size of this section is found in the `Size of the Debug Information` in the header section.

External Reference Definition Section

The format for the external reference section follows.

External References Count (2 bytes) (4 bytes—9.1)

The count indicates the number of references to external symbols that follow.

External References (variable)

Each external reference has the following format:

Name (1-n bytes)
Reference Count (2 bytes) (4 bytes—9.1)
References (reference count x 6 bytes):
Location Flag (2 bytes)
Reference Offset (4 bytes)

The reference count indicates the number of references to follow. Each reference comprises a location flag and a reference offset. The location flag specifies:

- The location of the reference (code, initialized data, or remote initialized data)
- The size of the reference (1, 2, or 4 byte)
- Whether the referenced value should be negated before being added to the value at the location
- Whether the referenced value should be made relative to the location

Bits 3-4	Size of reference to external symbol
01	1 byte
10	2 bytes
11	4 bytes

Bit 5	Reference Location
0	Specifies reference is in data area. (Bit 9 determines normal or remote.)
1	Specifies reference is in code area
Bit 6	Negative reference flag. If set, this tells the linker to add the negative of the symbols location when resolved.
Bit 7	Relative reference flag. If set, this tells the linker that the reference is relative to the location of the reference.
Bit 9	Data area section
0	Specifies non-remote
1	Specifies remote

Example values for the location flag include:

0x0008	1 byte initialized data
0x00b0	2 bytes, relative reference flag set, code
0x00b8	4 bytes, relative reference flag set, code
0x0218	4 bytes, initialized remote data
0x0248	1 byte, negative, initialized remote data

The reference offset is the offset into the code or initialized data section where the reference appears.

Local Reference Section

Local References Count (2 bytes) (4 bytes—9.1)

The count indicates the number of local references that follow.

Local References (variable)

These are references in the code or initialized data areas that reference code or data in the `psect`. Each local reference has the following format:

Type/Location flag (2 bytes)
 Local Offset (4 bytes)

The following type flags describe the location referred to by the reference.

Bits 0-2 determine the type:

Bit 2	0	Specifies data definition
1	1	Specifies code definition

Code type flags:

Bits 0-1	00	Code area
----------	----	-----------

Data type flags:

Bits 0-1	00	Uninitialized
	01	Initialized
	10	Uninitialized remote
	11	Initialized remote

Bit 81	Common block definition
--------	-------------------------

The following type flags describe the location where the reference appears. The following bits are defined:

Bits 3-4	Size of reference to external symbol
01	1 byte
10	2 bytes
11	4 bytes

Bit 5	0	Specifies data area
1	1	Specifies code area

Bit 6	Negative reference flag. If set, this tells the linker to add the negative of the symbols location when resolved.
-------	---

Bit 7	Relative reference flag. If set, this tells the linker that the reference is relative to the location of the reference.
Bit 90	Specifies non-remote 1 Specifies remote

Example type/location flag values include:

0x0008	1 byte from <code>init</code> data to data
0x0073	2 byte negative reference from code to <code>init</code> remote data
0x00e9	1 byte relative and negative from code to <code>init</code> data
0x0132	2 byte from code to remote common
0x01fa	4 byte relative and negative from code to remote common
0x0308	1 byte from <code>init</code> remote data to common
0x030a	1 byte from <code>init</code> remote data to remote common
0x0358	4 byte negative reference from <code>init</code> remote data to common
0x035a	4 byte negative reference from <code>init</code> remote data to remote common

ROF Edition Number 15 Format

ROF Edition Number 15 comprises these sections:

- **Header Section**
- **External Definition Section**
- **Object Code Section**
- **Initialized Data Section**
- **Remote Initialized Data Section**
- **Constant Data Section**
- **Remote Constant Data Section**
- **Debug Information Section**
- **Externally Referenced Symbol Data Section**
- **Expression Tree Data Section**
- **Reference Data Section**

Header Section

ROF Sync Bytes (4 bytes)

Sync bytes used by the linker to recognize an ROF.

Type/Language (2 bytes)

The type/language word from the `psect`. The linker uses the type and language to determine the desired module format. If this word is zero, the routine is assumed to be a subroutine type module. Only the root `psect` can have this word be non-zero.

Attribute/Revision (2 bytes)

Attribute/revision word to place in the module. It is only meaningful on a root `psect`.

Assembly Valid (2 bytes)

Word used to prevent the linker from linking erroneous modules. It is non-zero if assembly errors occurred.

Series (2 bytes)

Informs the linker of the assembler version used. This prevents problems that could occur in mixing different versions of the linker and the assembler.

Date/Time Assembled (6 bytes)

Indicates the date and time of assembly.

Edition number (2 bytes)

A user-definable edition number to place in the output module for root `psects`. For non-root `psects`, this word is informational only.

Size of Static Storage (4 bytes)

Informs the linker how much static data storage to reserve for the module. This value is determined from the total size of the `ds` directives in the `vsects`.

Size of Initialized Data (4 bytes)

Informs the linker how much initialized data is contained in the module. The size is determined by the total size of all `dc` directives in the `vsects`.

Size of Constant Data (4 bytes)

Reserved

Size of the Object Code (4 bytes)

This is determined from the size of the assembled code.

Size of Stack Required (4 bytes)

Informs the linker how much stack space the module requires. The value is obtained directly from the `psect` directive.

Offset to Entry Point (4 bytes)

Offset to the entry point in the object code. The offset is relative to the beginning of the module. The value is obtained directly from the `psect` directive.

Offset to Uninitialized Trap Entry Point (4 bytes)

Offset to the entry point in the object code which is called when a `tcall` is made without installing the appropriate trap handler. The offset is relative to the beginning of the module and is obtained directly from the `psect` directive.

Size of Remote Static Storage (4 bytes)

Informs the linker of the amount of remote static data storage to reserve for the module. This value is determined from the total size of the `ds` directives in the remote `vsects`.

Size of Remote Initialized Data (4 bytes)

Informs the linker of the amount of remote initialized data contained in the module. The size is determined by the total size of all `dc` directives in the remote `vsects`.

Size of Remote Constant Data (4 bytes)

Reserved

Size of the Debug Information (4 bytes)

This is determined from the size of the assembled debugger code.

Target Processor Type (2 bytes)

Informs the linker which target processor the ROF is intended to target. The high byte contains a code representing the processor family and the low byte contains a code specific to a processor within that family.

Code Information (2 bytes)

Information about the code in the file may be found here. For example, whether the code is thread-using or thread-safe.

Header Expansion (2 bytes)

Reserved

Name of Module (variable length)

Null terminated ASCII string taken directly from the `psect` directive. The linker uses the name to identify the `psect` in case of an unresolved reference or other error.

External Definition Section

External Definition Count (4 bytes)

Indicates the number of external definitions that follow. The external definitions are placed in the linker symbol table and can be referenced by any other module.

External Definitions (variable)

Each external definition has the following format:

Name (1-n bytes)
 Type Definition Flags (2 bytes)
 Symbol value (4 bytes)

Bits 0-2 and 8 determine the type:

Bit 20	Specifies <code>data</code> definition
1	Specifies <code>code</code> definition

If bit 2 is 0 (`data` definition flags):

Bits 0-1	00	Uninitialized non-remote
	01	Initialized non-remote
	10	Uninitialized remote
	11	Initialized remote

If bit 2 is 1 (`code` definition flags):

Bits 0-1	00	Code label
	01	<code>set</code> label
	10	<code>equ</code> label

Bit 8	0	Not a common block definition
1	1	Common block definition

Object Code Section

Object Code for the Module (variable length)

The size of this section is found in the `Size of Object Code` defined in the header section.

Initialized Data Section

Initialized Data (variable length)

The size of this section is found in the `Size of Initialized Data` defined in the header section.



Note

The object code section and/or the initialized data section may be omitted. If both are missing and the static data count is zero, the module can only contain absolute (`equ`) symbols. In this case, the linker extracts only those symbols that resolve an external reference. The `sys.l` library module is an example. It contains only `equ` symbol definitions.

Remote Initialized Data Section

Initialized Remote Data (variable length)

The size of this section is found in the `Size of Remote Static Storage` in the header section.

Constant Data Section

Constant Data (0 bytes)

Not used.

Remote Constant Data Section

Remote Constant Data (0 bytes)

Not used.

Debug Information Section

Debug Information (variable length)

The size of this section is found in the `Size of the Debug Information` in the header section.

Externally Referenced Symbol Data Section

Externally Reference Count (4 bytes)

Indicates the number of externally referenced symbol names that follow.

Externally Referenced Symbols(variable)

Each symbol name is null terminated.

Expression Tree Data Section

Expression Tree Count (4 bytes)

Indicates the number of expression tree structures that follow.

Expression Tree Structures(variable)

Each expression tree structure has the following format:

- Operator Code (2 bytes)
- First Operand (variable length)
- Second Operand (optional, variable length)

The operator code determines the type of first operand and the presence/type of the second operand.

Expression tree operands are of three types: constant operand, reference, or another expression tree.

Constant Operand (4 bytes)

The 4-byte value of the expression.

Reference (6 bytes)

Each reference has both a 2-byte reference flag and an associated 4-byte reference value.

A reference may be either a local or an external reference. The interpretation of the reference depends on the reference type.

In a local reference, the reference value is the offset to the local code or data object within this `psect`. This offset is used as the base value from which the expression return value is calculated.

In an external reference, the reference value is the index of the referenced symbol within the list of externally referenced symbols defined earlier. Numbering within this list begins with zero. The value of the referenced symbol is the offset used as a base for calculating the expression return value.

The relevant bits of the reference flag are defined as follows.

Bits 0-2 and 8 define what a local reference refers to:

Bit 20	Specifies <code>data</code> definition
1	Specifies <code>code</code> definition

If bit 2 is 0 (`data` definition flags):

Bits 0-1	00	Uninitialized non-remote
	01	Initialized non-remote
	10	Uninitialized remote
	11	Initialized remote

If bit 2 is 1 (`code` definition flags):

Bits 0-1	00	Code label
	01	<code>set</code> label
	10	<code>equ</code> label

Bit 8 0	Not a common block definition
1	Common block definition

Bits 3-4 specify the alignment required of an external symbol:

Bits 3-4	00	1 byte aligned
	01	2 byte aligned
	10	4 byte aligned
	11	8 byte aligned

Bit 7	Relative reference flag. If set, this tells the linker that the reference value is relative to the location of the reference.
-------	---

Bit 12 defines the type of the reference:

Bit 120 External reference

1 Local reference

Table 6-4 lists the operator codes defined in ROF Edition Number 15. The table also explains what the codes mean and gives the expected type of operands for each code.

Table 6-4 ROF Edition Number 15 Operator Codes

Operator Code	Operator Meaning	Type of Operand 1	Type of Operand 2
0x0000	constant expression	constant operand	none
0x0001	reference	reference operand	none
0x0002	hi (x) = x >>16	expression tree	none
0x0003	lo (x) = x & 0xffff	expression tree	none
0x0004	arithmetic negation (-)	expression tree	none
0x0005	bitwise negation (~)	expression tree	none
0x0006	bitwise and (&)	expression tree	expression tree
0x0007	bitwise or ()	expression tree	expression tree

Table 6-4 ROF Edition Number 15 Operator Codes (continued)

Operator Code	Operator Meaning	Type of Operand 1	Type of Operand 2
0x0008	bitwise xor (^)	expression tree	expression tree
0x0009	multiplication (*)	expression tree	expression tree
0x000a	division (/)	expression tree	expression tree
0x000b	addition (+)	expression tree	expression tree
0x000c	subtraction (-)	expression tree	expression tree
0x000d	left shift (<<)	expression tree	expression tree
0x000e	right shift (>>)	expression tree	expression tree
0x000f	arithmetic right shift	expression tree	expression tree
0x0010	$\text{high}(x) = (x \gg 16) + ((x \gg 15) \& 1)$	expression tree	none

Reference Data Section

Reference Count (4 bytes)

The number of reference structures that follow.

Reference Structures (variable)

References in the code or initialized data areas that reference code or data symbols. Each reference structure has the following format:

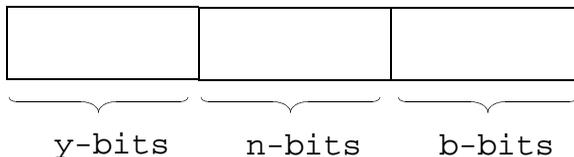
- Bit Number (1 byte)
- Field Length (1 byte)
- Location Flag (2 bytes)
- Local Offset (4 bytes)
- Expression Tree Index (4 bytes)

The expression tree index defines which index, from the previously defined list of expression trees, is to be evaluated. The resulting value is inserted in the bit field defined by the reference. The numbering of trees in the expression tree list starts with zero.

The local offset defines the location in the current `psect` where the linker needs to insert the bit field defined by this reference.

The reference bit field is defined by the bit number (b) and the field length (n). The bit field defined by these numbers, as related to the local offset, is defined in [Figure 6-5](#).

Figure 6-5 Bit Field Relationship to Local Offset



Where y , n , or b , are restricted by:

$$y < 8$$

$$0 < n < 32$$

$$b < 8$$

The location flag describes the location of the reference using the following flag bit definitions:

Bit 50	Specifies reference is in data area. (Bit 9 determines normal or remote.)
1	Specifies reference is in code area
Bit 90	Specifies non-remote
1	Specifies remote
Bit 100	Specifies reference bit field is unsigned
1	Specifies reference bit field is signed

Chapter 7: Assembler

The assembler translates an assembly language source file into a relocatable object file (ROF).



For More Information

Refer to [Chapter 6: Assembler and Object Code Linker Overview](#) for more information about ROFs.

This chapter covers the following topics:

- [Running the Assembler](#)
- [Symbolic Names](#)
- [Evaluating Expressions](#)
- [Input File Format](#)
- [Macros](#)
- [Directive Statements](#)
- [Pseudo-Instructions](#)



Running the Assembler

The assembler is a program that you can run from a command line interpreter or via `xcc`. Because the assembler name is different from processor to processor, this manual refers to the assembler as `<assembler>`. Replace `<assembler>` with the appropriate assembler name, such as `r68`, `a386`, or `appc`.



For More Information

Chapter 6: Assembler and Object Code Linker Overview, lists the valid assembler names.

The assembler command line syntax is:

```
<assembler> [<option(s)>] <file_name> [<option(s)>]
```

`<file_name>` is the name of the file to assemble.

`<option(s)>` is one or more options separated by spaces. If an option is not expressly specified, the assembler assumes a default condition for it.



For More Information

Refer to **Chapter 5: Compiler Phase Options** for a list and description of assembler options.



Note

An `opt` directive within the source program can override some options. Refer to the **Directive Statements** section in this chapter for more information about `opt`.

When a program is being assembled, the assembler does not recognize the addresses of names which are external references to other program sections. For example, a branch instruction to a label in another program section cannot have its offset computed because the address of the destination label is not known until the linker combines all sections. Therefore, when an external reference is encountered, the assembler sets up information in the ROF which identifies the instructions that reference external names. Because the assembler is not aware of what the actual offset within the module will be, each section is assembled as though it starts at offset 0.

By default, the assembler listing output is directed to the standard output path, which is usually the terminal display. To redirect the assembler output, use the shell output redirection modifier (`>`) on the command line.

The assembler automatically allocates memory for its working data area. It requests memory as needed up to the maximum available memory.

The following are typical assembler command lines. They are functionally identical, but the second command uses an alternative way of combining options.

```
<assembler> prog5 -l -s --c >listing  
<assembler> prog5 -ls --c >listing
```

In this example, the source program is read from the file `prog5`. The options `l` and `s` are turned on and `c` is turned off.

Symbolic Names

A symbolic name consists of the following characters:

- Alphanumerics (a-z, A-Z, 0-9)
- Underscore (_)
- At sign (@)
- Dollar sign (\$)
- Period (.)

The first character cannot be a digit, a dollar sign, or a period. A symbolic name may be preceded by the optional equal sign (=) label designator.

Following are examples of legal symbol name usage.

HERE	=there	SPL030	PGM_A
Q1020.1	t\$integer	L.123.X	a002@



Note

The assembler is case-sensitive, that is, the names `val_A` and `VAL_A` are considered different names.

Following are examples are illegal symbol names.

2move	Starts with a digit
lbl#123	Pound sign (#) is not a legal name character

Names are defined when first used as a label on an instruction or directive statement. They must be defined exactly one time in the program, with the exception of `set` labels. If a name is redefined (used as a label more than once), an error message prints on subsequent definitions.

If a symbolic name is used in an expression before it is defined, the assembler assumes the name is external to the `psect`. Information is recorded about the reference so the linker can adjust the operand accordingly. However, external names cannot appear in constant operand contexts for assembler directives.

Evaluating Expressions

Operands of many instructions and assembler directives can include numeric expressions in one or more places. The assembler can evaluate expressions of almost any complexity using a form similar to the algebraic notation used in programming languages such as C and BASIC.

Expressions consist of “operands” and “operators”.

- Operands are symbolic names or constants.
- Operators specify an arithmetic or logical function.

All assembler arithmetic uses long word (internally, 32 bit binary) signed or unsigned integers in the range of 0 to 4,294,967,295 for unsigned numbers, or -2,147,483,648 to +2,147,483,647 for signed numbers.

In some cases, the expression context may dictate a certain range value. For example, if the expression is used to represent a signed word operand, then the result must be in the range of -32,768 to 32,767. Some expression contexts may also require that the expression result have a particular alignment. For example, on some processors, branch targets must be word aligned. The assembler checks the value of each expression and reports an error if the expression is incorrectly aligned or out of range for the given context.

The assembler evaluates expressions from left-to-right using the algebraic order of operations (multiplications and divisions are performed before additions and subtractions). Parentheses may be used to change the natural order of evaluation.

Expression Operands

The following items may be used as operands within an expression.

- **Decimal Constants**

An optional minus sign (-) followed by one to twelve digits. For example:

```
100          3164765    -32767
-999999     0          12
```

- **Hexadecimal Constants**

A dollar sign (\$) or 0x followed by one to eight hexadecimal characters (0-9, A-F, or a-f). For example:

```
$ec00       $1000       0xFFFF
$3          $0300       0xDEADFACE
```

- **Binary Constants**

A percent sign (%) followed by one to thirty-two binary digits (0 or 1). For example:

```
%0101      %10101010  %1111000011110000
```

- **Character Constants**

A single character enclosed by single quotes (' '). For example:

```
'X'        'c'        '5'
```

- **Symbolic Names**

Any legal symbolic name.



Note

Not all assemblers allow the same flexibility of use of external symbols in expressions. Refer to the Assembler section in the appropriate processor chapter of the *Ultra C/C++ Processor Guide* for further information on evaluating expressions.

- Location Counter Symbols

The asterisk (*) and period (.) characters are special symbols that represent assembler internal location counters. The asterisk character represents the value of the current location counter. The location counter in use depends on the block the assembler is currently processing. If the current block is within a `psect` but not in a `vsect`, the asterisk (*) contains the value of the code location counter. If the current block is within a `vsect`, the asterisk (*) contains the value of the non-remote initialized data counter or the remote initialized data counter, as appropriate for the `vsect` `remote` directives.



For More Information

Chapter 6: Assembler and Object Code Linker Overview, contains more information about the `vsects` directive.

The asterisk (*) is often used in expressions to calculate distances. For example:

```
0000 0000 lbl_1:    dc.b    0,0,0,0,0,0,0,0
                00000000
                0000
0008 fff8 lbl_2    dc.w    lbl_1-*    ;distance from here to lbl_1
000a 0004 lbl_3    dc.w    lbl_5-*    ;distance from here to lbl_5
000c fffe lbl_4    dc.w    *-lbl_5    ;distance from lbl_5 to here
000e 000e lbl_5    dc.w    *-lbl_1    ;distance from lbl_1 to here
```

The period (.) represents the current value of the offset origin. The “offset org” is initialized by the `org` directive and is used by the `do` and `lo` directives.



For More Information

For more information, refer to the individual descriptions of the `do`, `lo`, and `org` directives in the **Pseudo-Instructions** section of this chapter.



Note

The expressions associated with the `com`, `do`, `ds`, `dz`, `if`, `lo`, `rept`, and `spc` statements and the `type`, `lang`, `attr rev`, `stack size`, and `psect` directives must evaluate to constant values. A relocatable result may change when the module is loaded or linked. Consequently, if a relocatable symbol is used in one of these expressions, it must be subtracted from another relocatable symbol so that the result is a constant.

Expression Operators

Operators used in expressions operate on one operand (negative and NOT) or on two operands (all others). **Table 7-1** shows the available operators. The operators are listed in the order in which they are evaluated relative to each other; that is, logical OR operations are performed before multiplications. Operators listed on the same line have identical precedence and are processed from left to right when they occur in the same expression.

Table 7-1 Expression Operators

Operator	Description
-	negative
^ or ~	logical NOT
&	logical AND
! or	logical OR

Table 7-1 Expression Operators (continued)

Operator	Description
<code>^</code>	logical XOR
<code>*</code>	multiplication
<code>/</code>	division
<code>+</code>	addition
<code>-</code>	subtraction
<code><<</code>	shift left
<code>>></code>	shift right

Logical operations are performed bitwise; that is, the logical function is performed bit-by-bit on each bit of the operands.

All expression operators assume signed operands. Subtraction and addition functions work on signed (two's complement) or unsigned numbers. Division by zero is reported as an error.

Expressions Involving External Symbols

An external symbol is a symbol whose value is not known when the program section is assembled. The actual values of external references must be inserted later when the program is linked.

The linker can resolve a limited number of expressions involving external references. The expressions that can be used with external references depend on the ROF format supported by a specific processor.



For More Information

Refer to **Chapter 6: Assembler and Object Code Linker Overview**, for more information about ROF formats.

Input File Format

The assembler reads the specified assembly source code file for its input. Each line in the file is a text string terminated by an end-of-line (return) character. The maximum line length is limited only by the amount of memory.

The assembler expects each line in the input file to have from one to four fields separated by spaces and/or tabs. The following fields may be used; every field is optional:

- a label field
- an operation field
- an operand field containing 0 or more operands depending on the operation
- a comment field

There are also two special cases:

- An asterisk (*) in the first character position indicates a comment line. The entire line is printed in the listing, but is not otherwise processed.
- Blank lines may also be included in the input, but are likewise ignored.

Label Fields

The label field begins in the first character position of the line. A label is a symbolic name used as an identifier. Some statements, such as `equ` and `set`, require labels. Other statements, such as `spc` and `ttl`, do not allow labels. Labels have the following characteristics.

- They must be a legal symbolic name.



For More Information

Refer to the **Symbolic Names** section in this chapter.

- They must be unique. A label can be defined only once in a program, except when used with the `set` directive.



For More Information

Refer to the **Directive Statements** section in this chapter for more information about the `set` and `equ` directives.

- They can be preceded by an optional equal sign as a label designator (`=`). This character is not part of the label so a label preceded by a label designator is not unique from an identical label without the designator.

If the line does not contain a label, the first character of the line must be a space or a tab. If the label is present, the assembler defines it as the address of the first byte of where the instruction's object code or data is assigned. The `set` and `equ` statements are exceptions because the labels for these statements are assigned the value of the operand field.

A label may either be known globally or locally:

- When a colon (`:`) follows the label, the name is known globally by all files that are linked together. This allows you to branch or jump to a location in another file. It also allows other files to refer to the data offset.
- If no colon (`:`) appears after the label, the label is only known in the `psect` where it is defined.



For More Information

psects and **vsects** are explained in **Chapter 6**.

External symbols are divided into external data symbols and external code symbols.

- External data symbols are located outside of a `psect` but within a `vsect`.
- External code symbols are located outside of both `psects` and `vsects`.

It is unnecessary to predefine external symbols, except for the `equ` and `set` statements located outside a `psect`. These symbols behave the same as symbols located within a `psect`.

Internal symbols are also divided into internal data symbols and internal code symbols.

- Internal data symbols are located within a `vsect` which is within a `psect`.
- Internal code symbols are located within a `psect`, but not within a `vsect`.

Labels within internal `vsects` are further distinguished as being initialized or uninitialized. Typically, this distinction is made based on the directive used to define the label (`ds` for uninitialized and `dz` or `dc` for initialized.) If an instruction or directive is not specified for a label in a `vsect`, the label type defaults to initialized. Whenever possible, `vsect` labels should be placed on the same line as the instruction or directive with which they are associated.



Note

Typically, code symbols are used in program counter relative addressing modes and data symbols are used in data area pointer relative addressing modes. Be careful to use the labels in the appropriate context.

The Operation Field

The operation field specifies the machine language instruction or assembler directive statement mnemonic name. It follows the label field by one or more spaces or tabs. The assembler accepts instruction mnemonic names in either uppercase or lowercase characters.

Instructions cause one or more bytes of object code to be generated, depending on the specific instruction and addressing mode. Some assembler directive statements, such as `dz` and `dc`, also cause object code area bytes to be generated.



For More Information

Refer to the Assembler/ Linker section of the appropriate processor chapter in the *Ultra C/C++ Processor Guide* for information concerning the operation field.

Operand Field

The operand field follows the operation field by at least one space or tab. Some instructions do not use an operand field; other instructions and assembler directives require an operand to specify an addressing mode, operand address, and/or parameters. Some require a combination of source and destination operands.



For More Information

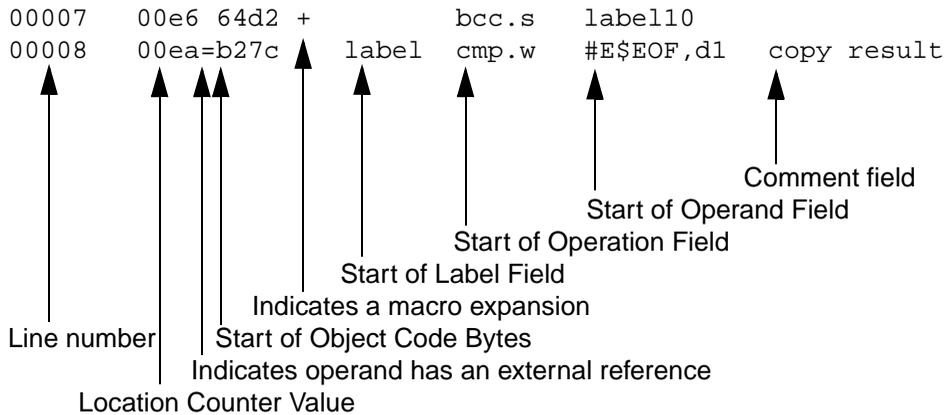
Refer to the specific instruction and assembler directive descriptions for the operand format in the Assembler/Linker section of the appropriate processor chapter in the *Ultra C/C++ Processor Guide*.

Comment Field

The last field of the source statement is the optional comment field. It can include a descriptive comment in the source statement. This field is not processed other than being copied to the program listing.

Assembly Listing Format

If you use the assembler's `-l` option, the assembler writes a formatted assembly listing to the standard output path. The output listing has the following format.



Macros

Identical or similar sequences of instructions may need be repeated in different places in a program. Writing a sequence of instructions repeatedly can be tedious if the sequence is long or must be used a number of times.

A macro defines an instruction sequence for use in more than one place within a program. A macro is given a name that is used similarly to any other instruction mnemonic. When the assembler encounters the name of a macro in the instruction field, it outputs all the instructions given in the macro definition. In effect, macros allow creation of new machine language instructions.

For example, suppose a program frequently performs left shifts of 64-bit quantities. This two-instruction sequence can be defined as a macro. For example:

```
dasl macro    * do a shift left
    asl.1 d1
    roxl.2 d0
endm
```

The `macro` and `endm` directives specify the beginning and the end of the macro definition, respectively. The label of the `macro` directive specifies the name of the macro. In this example, the name is `dasl`. When the assembler encounters the `dasl` macro, it can output code for `asl` and `roxl`. Normally, only the macro name is listed, but the assembler `-x` option may be used to list all instructions of the macro expansion.

Although macros are similar to subroutines, the distinction is:

- A macro repetitively duplicates an inline code sequence every time it is used and allows some alteration of the instruction operands.
- A subroutine appears exactly once and never changes. Subroutines are called using special processor specific branch instructions.

In those cases where macros and subroutines may be used interchangeably, macros usually produce longer but slightly faster programs. Short macros (6 bytes or less) are usually faster and shorter than subroutines because of the overhead of the needed branch instructions.

Macros can be an important and useful programming tool to significantly extend the assembler capabilities. In addition to creating instruction sequences, they may be used to create complex constant tables and data structures.



WARNING

When using macros, document them carefully. Macros can impair a program's readability if used indiscriminately and unnecessarily. This can make it extremely difficult to understand the program logic.

Structure

A macro definition consists of three sections:

```

<name> macro  the macro statement assigns a name
               to the macro
.
.
body          the macro body contains the macro
               statements
.
.
endm         the endm statement ends the macro
  
```

The label given in the `macro` statement defines the name of the macro. The name can be any legal assembler label. Instructions themselves may be redefined by defining macros having identical names.



WARNING

Redefining assembler directives, such as `ds`, can have unpredictable consequences.

The body of the macro can contain any number of legal assembler instructions or directive statements including references to previously defined macros.



Note

Rules for creating legal assembler labels are addressed in the **Symbolic Names** section in this chapter.

The last statement of a macro definition must be `endm`.

The assembler creates and maintains a temporary file to store the text of the macro definition. This file has a 1K buffer to minimize disk accesses. You should arrange programs that use more than 1K of macro storage space so that short, frequently used macros are defined first. This allows them to be kept in the memory buffer instead of disk space.

The body of a macro definition may contain a call to another macro. However, a macro cannot be defined within another macro.

Macro calls may be nested up to eight levels. For example, the following macro consists of two iterations of the `mac1` macro:

```
times2 macro
    mac1
    mac1
endm
```

Arguments

The way the assembler expands a macro may be changed by using arguments. For example, arguments may be used to specify operands, register names, constants, or variables in each occurrence of a macro.

A macro can have up to nine formal arguments in the operand fields. Each argument consists of a backslash character and the sequence number of the formal argument (\1, \2 ... \9). When the assembler expands the macro, it replaces each formal argument with the corresponding text string actual argument given in the macro call. Arguments may be used in any part of the operand field, but not in the instruction or label fields. Formal arguments may be used in any order and any number of times.

For example, the following macro performs a typical instruction sequence to write a buffer:

```
writ macro
    moveq    #\1,d0      * Get path
    moveq    #\2,d1      * Number of chars to write
    lea     \3(a6),a0    * Get address of buffer
    bsr     writbuff
endm
```

This macro uses three arguments:

- \1 for the path number
- \2 for the number of characters to write
- \3 for the address of the buffer

When `writ` is referenced, the assembler replaces each argument with the corresponding string given in the macro call. For example:

```
writ 1,2,Buf
```

This macro call is expanded to the code sequence:

```
moveq #1,d0
moveq #2,d1
lea   Buf(a6),a0
bsr  writbuff
```

If an argument string includes special characters, such as backslashes (\) or commas (,), the string must be enclosed in double quotes (" ").

An argument may be declared null by omitting all or some arguments in the macro call. This makes the corresponding argument an empty string so substitution does not occur when it is referenced.

Two special argument operators can be useful in constructing more complex macros:

- \Ln Returns the length of the actual argument, n, in bytes
- \# Returns the number of actual arguments passed in the given macro call

These operators are most commonly used with the assembler conditional assembly facilities to test the validity of arguments used in a macro call or to change the way a macro works according to the actual arguments used.

When macros perform error checking, they can report errors using the `fail` directive.



For More Information

The `fail` directive is described in the **Directive Statements** section of this chapter.

For example, the `writ` macro on the previous page could be expanded for error checking:

```
writ macro
  ifne \#-3          * Must have exactly three arguments
  fail writ: Must have three arguments
  endc
  moveq  #\1,d0      * Get path
  moveq  #\2,d1      * Number of chars to write
  lea   \3(a6),a0    * Get address of buffer
  bsr  writbuff
endm
```

Automatic Internal Labels

Sometimes it is necessary to use labels within a macro. If a macro containing a label is used more than once, unique label names need to be generated to avoid multiple definition errors. A backslash followed by an at sign (\@) appearing in a label within a macro expansion is replaced with a macro expansion serial number.

The macro expansion serial number is incremented each time the macro is expanded and is unique to that particular macro expansion.

Here is an example of a macro that uses unique labels:

```
test macro
    tst.b  stat(a6)
    beq.s  t\@a
    addq.l #1,count(a6)
t\@a
    endm
```

The macro expands as follows:

```
    tst.b  stat(a6)
    beq.s  t00001a
    addq.l #1,count(a6)
t00001a
```

The second expansion is:

```
    tst.b  stat(a6)
    beq.s  t00002a
    addq.l #1,count(a6)
t00002a
```



Note

\@ simply expands to a number. Observe proper syntax when constructing labels.

Directive Statements

Assembler directive statements give the assembler information that affects the assembly process. Usually, these statements do not cause code to be generated. Read the descriptions carefully. Different directives treat labels as required, optional, or prohibited.

This section contains information on the directive statements identified in **Table 7-2**.

Table 7-2 Directive Statements

Directive Statement	Description
<code>end</code>	End Program
<code>equ</code>	Assign Value to Symbolic Name
<code>fail</code>	Return Error Message
<code>if...else...endc</code>	Conditional Assembly
<code>macro...endm</code>	Macro Definition
<code>nam</code>	Rename Program
<code>opt</code>	Set Assembler Options
<code>pag</code>	Begin New Page in Listing
<code>psect</code>	Program Section
<code>rept ... endr</code>	Repeat Assembly Sequence
<code>set</code>	Assign Value to Symbolic Name
<code>spc</code>	Insert Blank Lines

Table 7-2 Directive Statements (continued)

Directive Statement	Description
ttl	Rename Listing Title
use	Use External File
vsect	Variable Section

endEnd Program

Syntax`end`**Description**

`end` indicate the end of a program. Its use is optional. If no `end` is present in the source file, `end` is assumed when an end-of-file condition occurs. The `end` statement does not have labels.

equAssign Value to Symbolic Name

Syntax

```
<label> equ <expression>
```

Description

`equ` assigns a value to a symbolic name (the label).

`<label>` may be any legal assembler label name. Syntax for legal assembler labels is described in this chapter.

`<expression>` is the value to assign to the label. It may be an expression, a name, or a constant.

You can use the `equ` directive in any program section. The `equ` statement label name cannot have been defined previously. The operand cannot include a name that has not yet been defined (as yet undefined names whose definitions also use undefined names). `equ` is normally used to define program symbolic constants, especially those used with instructions. Although the `set` directive is similar to `equ`, there are differences:

- Symbols defined by `equ` can be defined only once in the program.
- Symbols defined by `set` can be redefined again by subsequent `set` statements.

**WARNING**

`equ` cannot reference another `equ` that references an external name. For example:

```
joe equ moe  
moe equ external
```

See Also

[set](#)

Example

```
FIVE equ 5  
OFFSET equ address-base  
TRUE equ $FF  
FALSE equ 0
```

failReturn Error Message

Syntax

```
fail <textstring>
```

Description

`fail` forces an assembler error to be reported.

`<textstring>` is the error message that is processed in the same manner as assembler-generated error messages. Because the entire line following the `fail` keyword is assumed to be the error message, `fail` cannot have a comment field.

`fail` is most commonly used with conditional assembly directives that you set up to test for various illegal conditions, especially within macro definitions.

Example

```
ifeq maxval  
    fail maxval cannot be zero  
endc
```

if...else...endc

Conditional Assembly

Syntax

```
ifxx <expression>
    <statements>
[else]
    <statements>
endc
```

Description

The `ifxx` statements provide conditional assembly capabilities. This allows selective assembly of specific parts of a program depending on a variable or computed value. A single source file can then selectively generate multiple versions of a program.

The `ifxx` statement uses a symbolic name or an expression as an operand, and a comparison is made with the result. If the result of the comparison is true, statements following the `ifxx` statement are processed. Otherwise, the following statements are not processed until an `endc` (or `else`) statement is encountered.

For example, the following `ifeq` statement compares the value of its operand to zero:

```
ifeq switch
. * Assembled only if switch is 0
.
endc
```

The `else` statement allows the `ifxx` statement to explicitly select one of two program sections to assemble depending on the truth of the `ifxx` statement. Statements following the `else` statement are processed only if the result of the comparison is false. For example:

```

ifeq switch
. * Assembled only if switch is 0
.
else
. * Assembled only if switch is not 0
.
endc

```

The `endc` statement marks the end of a conditionally assembled program section.

Multiple `ifxx` statements may be used and may be nested within other `ifxx` statements. However, `ifxx` statements cannot have labels.

Each of the `ifxx` statements in [Table 7-3](#) perform a different comparison.

Table 7-3 Ifxx Statement Descriptions

Statement	Description
<code>ifeq</code>	True if operand equals zero
<code>ifne</code>	True if operand does not equal zero
<code>iflt</code>	True if operand is less than zero
<code>ifle</code>	True if operand is less than or equal to zero
<code>ifgt</code>	True if operand is greater than zero
<code>ifge</code>	True if operand is greater than or equal to zero
<code>ifdef</code>	True only if the specified symbol is defined
<code>ifndef</code>	True only if the specified symbol is not defined

`ifxx` statements that test for less than or greater than zero may be used to test the relative value of two symbols if the symbols are subtracted in the operand expression. For example, the following statement is true if `min` is greater than `max` (the statement literally means `if max-min < 0`):

```
iflt    max-min
```

The `ifdef` and `ifndef` directives are different from the other conditional assembly instructions because their operand field is a single label rather than an expression.

- For `ifdef`, if the specified symbol has been defined, the instructions within the conditional are assembled.
- For `ifndef`, if the symbol has not been defined, the conditional is assembled.

A symbol is considered to be defined if it appears in the label field before the reference during the first pass.



WARNING

Conditionals based on undefined (but to be defined) values cause phasing errors. When writing conditional assembly, ensure that the conditional evaluates to the same value during the first and second pass or phasing errors may result.

The `ifdef` and `ifndef` directives are useful for assembling sections of code based on the presence of a symbol. `ifdef` and `ifndef` are most useful when symbols are defined on the command line. This allows makefiles to pass symbols affecting the assembly without actually changing any definitions in a file.

macro...endm

Macro Definition

Syntax

```
<name> macro  
    <body of macro>  
endm
```

Description

`macro` defines an instruction sequence that may be used in more than one place within a program. A macro definition consists of three sections:

The `macro` statement This assigns a name to the macro. The name can be any legal assembler label. Syntax for legal assembler labels are provided in this chapter.

The body of the macro The body can contain any number of legal assembler instructions or directive statements, including references to previously defined macros.

The `endm` statement `endm` ends the macro.

The assembler creates and maintains a temporary file to store the text of the macro definition. This file has a 1K buffer to minimize disk accesses. Programs that use more than 1K of macro storage space should be arranged such that short, frequently used macros are defined first. This allows them to be kept in the memory buffer instead of using disk space.



For More Information

Macros are covered in greater detail in the **Macros** section of this chapter.

namRename Program

Syntax

```
nam <string>
```

Description

`nam` specifies the program name that is printed in a program listing. `nam` cannot have a label or a comment field.

The program name is printed on the left side of the second line of each listing page, followed by a dash, and then by the title line. The name and the title may be changed as often as desired.

See Also

[ttl](#)

Example

```
nam Datac  
ttl Data Acquisition System
```

Generates:

```
Datac - Data Acquisition System
```

Syntax

opt <option>

Description

opt sets or resets any of several assembler control options. Options are denoted by a single character. A preceding hyphen (-) turns the specified option off. One exception is the d option which must be followed by a number. opt must not have label or comment fields. The other exception is o which is followed by a path name.

Options

[-]c	Print a listing of conditional assembly lines in an assembler listing. This is off by default.
d <num>	Set the number of lines per page to <num> for a listing. The default is 66.
[-]e	Print errors. This is on by default.
[-]f	Use form feed instead of line feeds for page ejects. Use form feed for top of form. This is off by default.
[-]g	List all code bytes generated. This is off by default.
[-]h	Print information about probable hazards (MIPS only).
[-]i	Revert to original error message format.
[-]k	Force the assembler to keep the output file, even if there were errors generating it. This only applies if -o is used to create an output file.
[-]l	Write a formatted assembler listing to standard output. If not used, only error messages are printed. This is off by default.

<code>mt</code>	Mark ROF as thread-safe, thread-using.
<code>mts</code>	Mark ROF as thread-safe, non-thread-using.
<code>m <n></code>	Set target number (MIPS, SH-3, and SH-4 only).
<code>[-]n</code>	Omit line numbers from the assembler listing allowing more room for comments.
<code>o=<path></code>	Write the relocatable output to the specified file.
<code>p <n></code>	Align all orgs to <n>-byte boundary, <n> is 2, 4, 8, or 16.
<code>[-]q</code>	Quiet mode. Suppress warnings and nonfatal messages.
<code>[-]s</code>	Print the entire symbol table at the end of the assembly listing. This is off by default.
<code>[-]v</code>	Show version information.
<code>w <width></code>	Set the line width for the listing.
<code>[-]x</code>	Print the macro expansion in the assembler listing. This is off by default.
<code>z=<path></code>	Read addition options from <path>



For More Information

Additional system-specific options are defined in the *Ultra C/C++ Processor Guide*.

pag

Begin New Page in Listing

Syntax

`pag[e]`

Description

`pag` causes the assembler to begin a new page of the listing. It is used to improve the readability of program listings, and it is not printed.

`pag` cannot have a label.

See Also

[spc](#)

Syntax

```
psect [<name>,<typelang>,<attrev>,<edition>,  
      <stacksize>,<entrypt>[,<trapent>]]  
      <body>  
ends[ect]
```

Description

`psect` specifies the program code section. There can only be one `psect` per source file. The `psect` directive initializes all assembler location counters and marks the start of the program segment. You must declare all instruction statements and `vsect` data reservations within the `psect .. endsect` block.

`<name>` specifies a name the linker uses to identify the `psect`. Any printable character may be used except a space or comma. However, the name must begin with a non-numeric character. The name need not be unique from other `psect` names but it is easier to identify problem `psects` if the names are different.

`<typelang>` is a 16-bit expression to use as the executable module type/language word. If the `psect` is not a root `psect`, `typelang` must be zero.

`<attrev>` is a 16-bit expression to use as the executable module attribute/revision word.

`<edition>` is a 16-bit expression to use as the executable module edition word.

`<stacksize>` is a 32-bit expression that estimates the amount of stack storage required by this `psect`. The linker totals the value in all `psects` to appear in the executable module and adds the value to any stack storage requirement for the entire program.

`<entrypt>` is a 32-bit expression for use as the program entry point offset for the `psect`. If the `psect` is not a root `psect`, this should be 0.

<trapent> is a 32-bit expression indicating the uninitialized trap entry point offset. This is used for handling user-mode trap instruction processing. Only give this parameter if the program includes code to handle uninitialized traps otherwise, omit this parameter; do not use zero. This parameter is used only in root `psects`.

The `psect` may have a parameter list containing a name followed by five or six expressions if the `psect` is to be a root `psect`, or it can have no parameter list at all. If a parameter list is provided, it is stored in the ROF for later use by the linker to generate the memory module header. If a parameter list is not provided, the `psect` name defaults to `program` and all other parameters have default values of zero.

The following statements are legal within `psects`:

Any instruction mnemonic

```
align
dc
dz
ends
endsect
tcall
vsect
```



WARNING

`ds` may not be used within a `psect`.

See Also

[vsect](#)

rept ... endrRepeat Assembly Sequence

Syntax

```
rept <expr>
    <statements>
endr
```

Description

`rept` repeats the assembly of a sequence of instructions a specified number of times. The result of the operand expression is used as the repeat count. The expression cannot include external or undefined symbols.

`rept` loops may not be nested.

Example

```
* delay
    rept 10
    nop
    endr
```

setAssign Value to Symbolic Name

Syntax

```
<label> set <expression>
```

Description

`set` assigns a value to a symbolic name (the label).

`<label>` may be any legal assembler label. Syntax for the legal assembler labels is described in this chapter.

`<expression>` is the value to assign to the label. It may be an expression, a name, or a constant.

The `set` directive may be used in any program section. `set` is usually used for symbols used to control the assembler operations, especially conditional assembly and listing control. Although the `equ` directive is similar to `set`, the differences between the `equ` and `set` statement are:

- Symbols defined by `equ` can be defined only once in the program
- Symbols defined by `set` can be redefined again by subsequent `set` statements

**WARNING**

`set` cannot reference external names.

See Also

[equ](#)

Example

```
SUBSET set TRUE
  ifne SUBSET
    use subset.defs
  else
    use full.defs
  endc
SUBSET set FALSE
```

spc

Insert Blank Lines

Syntax

`spc <expression>`

Description

`spc` puts blank lines in the listing to improve the readability of program listings, and it is not printed.

`spc` cannot have a label.

`<expression>` determines the number of blank lines to generate. It must be a numeric constant. If no `<expression>` is given, a single blank line is generated.

See Also

[pag](#)

Syntax

```
ttl <string>
```

Description

`ttl` specifies the title line that is printed in a program listing. It cannot have a label or a comment field.

The program name is printed on the left side of the second line of each listing page, followed by a dash, and then by the title line. The name and title may be changed as often as desired.

See Also

[nam](#)

Example

```
nam Datac  
ttl Data Acquisition System
```

Generates:

```
Datac - Data Acquisition System
```

use**Use External File**

Syntax

```
use pathlist
use "pathlist"
use <pathlist>
```

Description

`use` temporarily stops the assembler from reading the current input file. It then requests the operating system to open the specified pathlist, from which input lines are read until an end-of-file occurs. At that point, the latest file is closed and the assembler resumes reading the previous file from the statement following the `use` statement.

`pathlist` is the path to the new input file.

- If `pathlist` is listed by itself or enclosed in quotation marks, the assembler searches for the file relative to the directory where the source file is located. For example, if the file name is listed by itself, the assembler looks in the directory holding the source file. If a relative path is specified (`..\..\filename`), the assembler looks in the directory specified by the relative path.
- If the pathlist is enclosed in angle brackets, the assembler searches for the file in the default include directories and any directories listed by the user with the `-v` or `-asu` option.

`use` statements may be nested (that is, a file being read due to a `use` statement can also perform `use` statements) up to the number of simultaneously open files that the operating system allows (usually 29 not including the standard I/O paths). Full or relative pathlists may be specified. Relative pathlists are relative to the current data directory.



Note

The default `use` directive search directory is `/mwoS /<OS> /SRC /DEFS`, where `<OS>` is `OS9` for 68K targets and `OS9000` for all others.

Syntax

```
vsect [remote]
```

Description

`vsect` specifies the variable storage section containing either initialized or uninitialized variable storage definitions. The `vsect` also specifies how the variables are intended to be addressed, normally or remotely. The assembler does not check that variables are addressed as they are declared. `vsect` causes the assembler to change the location counter from the code location counter to the data location counter. The data location counter that is used depends on the statement used and the presence of the word `remote` after the `vsect` directive.

There are four data location counters. One for each of the following types of data:

- Initialized
- Uninitialized
- Remote initialized
- Remote uninitialized

The following are legal internal statements:

```
align  
dc  
ds  
dz  
ends  
endsect
```

When `ds` is used, the uninitialized data location counter is used. When a `vsect remote` is in effect, the `ds` applies to the remote uninitialized data location counter.

The `dc` and `dz` directives set initial data values. The assembler uses the initialized data location counter for these directives. The constants appear in the data area of the program when executed. These values may then be modified.

The `dc` and `dz` directives can also appear outside of a `vsect` in the body of the `psect`. In this case, the constants are assembled into the program's code area. Do not change constants defined in this manner. To do so would make the program self-modifying and non-re-entrant.

The data location counters maintain values from one `vsect` block to the next. Since the linker handles the actual data allocation, there is not a facility to adjust the data location counters.



Note

The data location counters maintain values from one `vsect` block to the next. Since the linker handles the actual data allocation, there is not a facility to adjust the data location counters.



For More Information

Refer to the appropriate processor chapter in the *Ultra C/C++ Processor Guide* for information concerning the `vsect` directive.

See Also

`psect`

Pseudo-Instructions

Pseudo-instructions are special assembler statements that generate object code but do not correspond to actual machine instructions. Their primary purpose is to create special sequences of code and/or constant data to be included in the program. Labels are optional on pseudo-instructions.



For More Information

The processor-specific chapters in the *Ultra C/C++ Processor Guide* may contain additional pseudo-instructions.

This section comprises information on pseudo-instructions identified in [Table 7-4](#).

Table 7-4 Pseudo-Instructions

Pseudo-Instruction	Description
<code>align</code>	Align to a Specific Boundary
<code>com</code>	Reserve Memory for Common Block
<code>dc</code>	Define Constant
<code>do</code>	Assign Offset Counter Value to Label
<code>ds</code>	Define Storage
<code>dz</code>	Reserve Zero Bytes
<code>lo</code>	Decrement Offset Counter, Then Assign Value to Label

Table 7-4 Pseudo-Instructions (continued)

Pseudo-Instruction	Description
<code>org</code>	Set Offset Counter Origin
<code>tcall</code>	Invoke trap handler

align

Align to a Specific Boundary

Syntax

```
align <alignment boundary>
```

Description

`align` aligns the next generated code or next assigned data offset on some byte boundary in memory. If the current value of the instruction counter is not aligned to the alignment boundary, sufficient zero bytes are inserted in the object code to force the desired alignment.

`<alignment boundary>` specifies the alignment to use. If `<alignment boundary>` is not specified, `align` uses an alignment boundary of two bytes. `<alignment boundary>` must be a power of two.

If `align` is specified in a `vsect`, the assembler aligns both the initialized and uninitialized location counters.

`align` is generally used after odd length constant tables, character strings, or character data are embedded in the object code.

See Also

[dc](#)

[ds](#)

Syntax

```
<label>: com.<s> <size>
```

Description

`com` reserves an area of memory in the appropriate `vsect` for use as an overlaid common block. The size of the data area actually assigned by the linker is the maximum of the sizes given on all `com` statements for that label.

`<label>` is any legal label name. The label can appear in any number of `psects`.

The `size` extension, `.s`, can be:

- `.b` for bytes
- `.w` for words (default)
- `.l` for longwords

To facilitate initialization of common blocks, the label is allowed to appear on initialized data directives. In this case, the data definition is used instead of the size given on the `com` statement.

`com` may be used in a `vsect` remote to allocate a common block in the remote memory area.

Example

The following allocates a non-remote data common block of 100 bytes. Both references to `block1(a6)` in each file refer to the same address.

File t1.a

```
    vsect
block1:  com      100
        ends
        clr.b    block1(a6)
```

File t2.a

```
vsect
block1:  com    100
        ends
        clr.b  block1(a6)
```

The following example demonstrates how to initialize a common block. The first `block2 com` directive reserves 12 bytes of memory. The `block2` definition in `s2.a` defines initialized data for the common area. The initializing data definitions supersede the sizes given on any `com` directive. For best results, the sizes of the `com` directives and the amount of initializing data should agree.

File s1.a

```
vsect
block2:  com    12
        ends
        move.l  block2+8(a6),d0
```

File s2.a

```
vsect
block2:  dc.l   1,2,3
        ends
```

Syntax

```
<label> dc.<size> <expression>{ ,<expression>}
```

Description

`dc` generates sequences of one or more constants (initialized data) of various sizes within the program. The argument is a list of one or more expressions or character strings. If more than one expression or string is used, they are separated by commas.

The `.<size>` extension can be any of the below sizes:

- `.b` for bytes
- `.sb` for signed bytes
- `.w` for words (16-bit, default)
- `.sw` for signed words (16-bit)
- `.l` for longwords (32-bit)
- `.sl` for signed long words (32-bit)

The signed variants give the linker additional information about the nature of an reference. For example, if a signed word (`.sw`) external reference appeared in one psect and another psect defined the value as `0xf000`. The linker would complain about the value `0xf000` being too large to express in a 16-bit signed field. The linker would not have complained if plain `.w` were used on the external reference. Use the signed variants to ensure that large positive values are not accidentally interpreted as negative values.

A `dc` used in a `vsect` creates an initialized data variable (read/write) in the process' data area. The initialization value is stored in a special section of the object code and is copied to the appropriate locations in the data area by the operating system.

A `dc` located outside a `vsect` creates read-only constants in the program area. The program should not change these constants.

Character string constants can be any sequence of printable ASCII characters enclosed in double quotes. For `dc.w` and `dc.l`, a string constant is padded with zeroes on the right end if it does not fill the final word or long word. Therefore, `dc.b` is the most natural format for strings.



Note

`dc.w` and `dc.l` automatically align to an even-byte boundary if the respective location counter is not on an even-byte boundary.

Example

```
dc.b 1,20,"A"  
dc.b index/2+1,0,0,1  
dc.w 1,10,100,1000,10000  
dc.w $F900,$FA00,$FB00,$FC00  
dc.b "most programmers are strange people",0  
dc.b "0123456789"
```

Syntax

```
<label> do.<size> <expression>
```

Description

`do` assigns an increasing set of values to a set of symbolic names. It is unrelated to memory allocation.

The `.<size>` extension can be:

- `.b` for bytes
- `.sb` for signed bytes
- `.w` for words (16-bit, default)
- `.sw` for words (16-bit)
- `.l` for longwords (32-bit)
- `.sl` for signed longwords (32-bit)

`do` and `lo` provide a convenient means of defining a group of names with sequentially related values. Some examples are error codes, character sets, and stacked variables.

Each time a `do` is encountered, its label is assigned the current value of the offset counter. The offset counter is then incremented by the result of the expression multiplied by:

- 1 for a byte
- 2 for a word
- 4 for a long

See Also

[lo](#)

[org](#)

Example

```

org $500
joe do.l 1* is the same as joe equ 500
moe do.l 1* is the same as moe equ 504

```

If an offset needs to be assigned to more than one label, the set pseudo opcode may be used. For example:

```

org 'A'
A          do.b 1      * Gives label A the value of its ASCII code
B          do.b 1      * Gives label B the value of its ASCII code
C          do.b 1      * Gives label C the value of its ASCII code
D          do.b 1      * Gives label D the value of its ASCII code
E          do.b 1      * Gives label E the value of its ASCII code

ret_pc     do.l 1      * pushed return PC
reg_space  set  ret_pc * amount of stack space required for
              * saved registers

```

Syntax

```
<label> ds.<size> <expr>
```

Description

`ds` is used within a `vsect` to declare storage for uninitialized variables in the data area.

The `.<size>` extension can be:

- `.b` for bytes
- `.sb` for signed bytes
- `.w` for words (16-bit, default)
- `.sw` for words (16-bit)
- `.l` for longwords (32-bit)
- `.sl` for signed longwords (32-bit)

`<expr>` specifies the size of the variable in bytes, words, or longwords depending on the size given for the `ds` extension. This value is added to the appropriate uninitialized data location counter in order to update it.

When `ds` is used to declare variables, a label is usually specified which is assigned the variable's relative address. In OS-9 for 68K and OS-9, the address is not absolute. Instead, indexed addressing modes are used to access variables. The actual relative address is not assigned until the linker processes the ROF.



Note

`ds.w` and `ds.l` automatically align to an even byte boundary if the respective location counter is not even.

Syntax

```
<label> dz.<size> <expression>
```

Description

`dz` fills memory with a sequence of bytes, each having a value of zero.

The `.<size>` extension can be:

- `.b` for bytes
- `.sb` for signed bytes
- `.w` for words (16-bit, default)
- `.sw` for words (16-bit)
- `.l` for longwords (32-bit)
- `.sl` for signed longwords (32-bit)

`<expression>` is used as the number of zero values to place in the appropriate code or initialized data section. A `dz` used within a `vsect` is considered initialized data that the program can alter.

When using OS-9, do not reserve zero bytes in the initialized data area (`vsect`). The operating system automatically zeroes the data area. Therefore, a `dz` in the `vsect` only wastes space in the module.

A `dz` used within a `psect` creates a read-only zero constant that the program should not change.

Example

```
dz.b 24 * Reserve 24 zero value bytes
dz.w 1  * Reserve 1 zero value word
```

Decrement Offset Counter, Then Assign Value to Label

Syntax

```
<label> lo.<size> <expression>
```

Description

lo assigns a decreasing set of values to a set of symbolic names. It is unrelated to memory allocation.

The .<size> extension can be:

- .b for bytes
- .sb for signed bytes
- .w for words (16-bit, default)
- .sw for words (16-bit)
- .l for longwords (32-bit)
- .sl for signed longwords (32-bit)

do and lo provide a convenient means of defining a group of names with sequentially related values. Some examples are error codes, character sets, and stacked variables.

When an lo statement is encountered, the offset counter is decremented by the appropriate size and the result is assigned to its label. This is useful when used with the stack frame operations.

See Also

[do](#)

[org](#)

Example

```
org $500
joe lo.l 1      * is the same as joe equ $4FC
moe lo.l 1     * is the same as moe equ $4F8
org 'Z'+1
Z    lo.b 1    * gives label Z the value of its ASCII code
```

Y	l o . b 1	* gives label Y the value of its ASCII code
X	l o . b 1	* gives label X the value of its ASCII code
W	l o . b 1	* gives label W the value of its ASCII code
V	l o . b 1	* gives label V the value of its ASCII code

Syntax

```
org <expression>
```

Description

`org` sets or changes the origin (starting value) of the offset counter.

See Also

[do](#)

[lo](#)

Example

```
org $500
joe do.l 1      * is the same as joe equ 500
moe do.l 1      * is the same as moe equ 504
org 'A'
A    do.b 1     * gives label A the value of its ASCII code
B    do.b 1     * gives label B the value of its ASCII code
C    do.b 1     * gives label C the value of its ASCII code
D    do.b 1     * gives label D the value of its ASCII code
E    do.b 1     * gives label E the value of its ASCII code
```

tcall

Invoke trap handler

Syntax

```
tcall <vector>,<function>
```

Description

The `tcall` built-in macro generates user trap calls. User traps access trap handlers. `tcall` has two arguments, a vector number (zero through 15) and a function code. The code generated for `tcall` varies from processor to processor, refer to the appropriate chapter in ***Ultra C/C++ Processor Guide*** for more information.

Chapter 8: Prelinker

This chapter includes the following sections:

- **Automatic Instantiation**
- **Prelinker Execution**
- **Creating C++ Libraries Containing Templates**



Automatic Instantiation

The goal of an automatic instantiation mode is to provide seamless instantiation. The processes of compiling source files to object code, linking object code, and running the resulting program should occur without concern how the necessary instantiations are performed.

In practice, seamless instantiation is difficult for a compiler. Compilers use different automatic instantiation schemes with different strengths and weaknesses.

Ultra C/C++ requires, for each instantiation required, some source file (normal, top-level, explicitly-compiled) that contains both the definition of the template entity and types required for the particular instantiation.



Note

This is not always the case. Suppose that file A contains a definition of class X and a reference to `Stack<X>::push`, and that file B contains the definition for the member function `push`. There would be no file containing both the definition of `push` and the definition of X.

This requirement can be met in various ways:

- Each header file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.
- Implicit inclusion: when the compiler encounters a template declaration in a header file and discovers a need to instantiate that entity, it receives permission to look for an associated definition file having the same base name and a different suffix, and it implicitly includes that file at the end of the compilation. This method enables Ultra C/C++ compilation of most programs written using the `cfront`.



For More Information

Reference the **Implicit Inclusion** section in **Chapter 12: Language Features**.

- The ad hoc approach: the programmer ensures that the files defining template entities also comprises the definitions of the available types, and adds code or pragmas in those files to request instantiation of the entities there.

The Ultra C/C++ automatic instantiation method works as follows:

1. The first time the source files of a program are compiled, template entities are not instantiated. However, the generated object files contain information about things that could have been instantiated in each compilation. For any source file that makes use of a template instantiation an associated `.ii` file is created if one does not already exist (the compilation of `abc.C` would result in the creation of `abc.ii`).
2. When the object files are linked, a program called the prelinker is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.
3. If the prelinker finds a reference to a template entity for which there is not a definition in the set of object files, it looks for a file that indicates that it could instantiate that template entity. When it finds the appropriate file, it assigns the instantiation to it. The set of instantiations assigned to a given file is recorded in the associated `.ii` file.
4. The prelinker then executes the compiler again to recompile each file for which the `.ii` file was changed.
5. When the compiler compiles a file, it reads the `.ii` file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and all the other things that were already in the object file).

6. The prelinker repeats steps 3 through 5 until there are no further instantiations to be adjusted.
7. The object files are linked.

Once the program is correctly linked, the `.ii` files contain a complete set of instantiation assignments. Further recompilation of source files causes the compiler to reference the `.ii` files and perform the indicated instantiations as it performs the normal compilations. Except in cases where the set of required instantiations changes, the prelink step determines that all the necessary instantiations are present in the object files and that instantiation assignment adjustments are unnecessary (true even if the entire program is recompiled).

If the programmer provides a specialization of a template entity in a program, the specialization is seen as a definition by the prelinker. That definition satisfies whatever references there might be to that entity, so the prelinker determines that there it is unnecessary to request an instantiation of the entity. If the programmer adds a specialization to a program previously compiled, the prelinker removes the assignment of the instantiation from the proper `.ii` file.

The `.ii` files should not, in general, require manual intervention. One exception: if a definition is changed such that some instantiation no longer compiles (generates an error), and at the same time a specialization is added in another file, and the first file is recompiled before the specialization file and errors are generated, the `.ii` file for the file generating the error must be deleted manually to enable the prelinker to regenerate it.

If the prelinker changes an instantiation assignment, a message is issued:

```
C++ prelinker: A<int>::f() assigned to file test.o  
C++ prelinker: executing: xcc -eas -fd=test.r test.c
```

The automatic instantiation scheme can coexist with partial explicit control of instantiation by the programmer through the use of pragmas or command line specification of the instantiation mode.

Prelinker

The prelinker manages link-time automatic instantiation of template entities in C++ programs. It is usually run prior to the object code or I-code link step by the executive. Based on the information contained in the object code or the I-code files, the prelinker may recompile the sources that generated the object code or the I-code files. After the prelinker finishes, all needed template entities are present in the object code or I-code files.

Prelinker Execution

The syntax for calling the prelinker is:

```
prelink [<options>] <files>
```

Usually, options are not necessary.



For More Information

Refer to [Chapter 5](#) for a description of options that may be passed to the prelinker.

The list of files passed to the prelinker can be either a combination of ROFs and libraries (created using `libgen`) or a combination of I-code files and I-code libraries (created using `ilink`).

Description and Example

Prelinker execution is illustrated by the following example C++ program.

Consider a file `stack.h` containing the following template definition:

```
template<class T> class stack {
public:  stack() { /* ... */ } // inline
        ~stack() { /* ... */ } // inline
        void push(T elem);
        T pop();
};

template<class T>
void stack<T>::push(T elem)
{ /*...*/ }
template <class T>
T stack<T>::pop()
{ /*...*/ }
```

Also, consider two files, `stack1.cpp` and `stack2.cpp`, with the following content:

`stack1.cpp`

```
#include "stack.h"
//...
void f()
{
    stack<int>  istack;
    istack.push(0);
    istack.push(1);
    istack.push(istack.pop() + istack.pop());
}
```

`stack2.cpp`

```
#include "stack.h"
//...
void g()
{
    stack<int>  istack;
    istack.push(0);
    stack<char> cstack;
    cstack.push('a');
}
```

Upon compilation of each file to a ROF, data and code in each ROF may be inspected using the command:

```
libgen -ln <ROF> ! decode
```

Note that `stack1.r` contains **can-be-instantiated** and **template-instantiation-request** flags for:

```
int stack<int>::pop()
void stack<int>::push(int)
```

Since the constructor and destructor of class `stack` were inlined, these were instantiated but the member functions `push(int)` and `pop()` were not; only requests for instantiation were registered. This is necessary since the compiler has compiled only part of the program and acts to avoid multiple symbol definitions that would arise if all referenced template entities were instantiated.

`stack2.r` contains **can-be-instantiated** flags for:

```
int stack<int>::pop()
void stack<int>::push(int)
char stack<char>::pop()
void stack<char>::push(char)
```

and **template-instantiation-request** flags for

```
void stack<char>::push(char)
void stack<int>::push(int)
```

Again, note that though there are **can-be-instantiated** flags for all templated entities, instantiation requests were registered only for those that are actually referenced.

The instantiation-information files `stack1.ii` and `stack2.ii` at this point contain just the compiler command lines used to create the ROFs, the file name, and the name of the directory where the file resides. This information is used by the prelinker in the next step to recompile the source files to produce definitions for all referenced symbols.

When linking the two ROFs together using the command line,

```
xcc -qp stack1.r stack2.r
```

the compiler calls the prelinker as

```
prelink stack1.r stack2.r <libraries>
```

Ignoring `libraries` on the prelinker command line for this particular example, the following output is produced:

```
C++ prelinker: void stack<T1>::push(T1) [with T1=int] assigned
to file stack1.r
C++ prelinker: T1 stack<T1>::pop() [with T1=int] assigned to
file stack1.r
C++ prelinker: void stack<T1>::push(T1) [with T1=char] assigned
to file stack2.r
C++ prelinker: executing: xcc -eas -fd=stack1.r stack1.cpp
```

```
C++ prelinker: executing: xcc -eas -fd=stack2.r stack2.cpp
```

The `stack1.i` file now contains the (mangled) names of the template instantiations assigned to `stack1.r`:

```
int stack<int>::pop()  
void stack<int>::push(int)
```

In the same way, `stack2.i` contains the (mangled) names of the template instantiations assigned to `stack2.r`:

```
void stack<char>::push(char)
```

Although both `stack1.r` and `stack2.r` referenced `stack<int>::push(int)`, it was assigned to only one ROF (`stack1.r`), eliminating multiple definitions. No instance was generated for `stack<char>::pop()` as it was not referenced in any of the ROFs.

During the next compilation and linking, neither instantiation assignments nor recompilation occur. This is because the compiler uses the information in the `.i` files to perform all the needed instantiations in the first compile.

The prelinker can also be called on I-code files if I-code linking is used. In this case, `prelink` is called with a `-filink` option and a list of I-code files and libraries. The execution is similar to the prelinker description and example provided for ROFs in this section.

Creating C++ Libraries Containing Templates

This section discusses the following Ultra C++ topics:

1. **Creating Libraries which Reference Templated Entities**
2. **Building C++ Libraries Referencing Template Entities**
3. **Structuring and Building Template Libraries**

The above topics are not completely independent: topic (2) may imply topic (1).

Terms and Definitions

- An **instantiation** of a class or function template is generation of an instance of the template for a particular type or types (which form the arguments of the template).
- A **templated entity** is a class or function template which may or may not have been instantiated for a specific type or types before it is referenced; it just needs to be declared before being referenced.
- A **template specialization** is a user-supplied instantiation of a class or a function template that provides "special case" instantiation.
- An **explicit instantiation** directive causes the compiler to generate instantiations for class or function templates irrespective of whether these are actually referenced or not.
- A **template library** is a collection of class or function templates in source form, present in header files; part of the library may be in compiled form (object-code or i-code). Usually, it is not possible to obtain the compiled portion just from the source in the header files.

Creating Libraries which Reference Templated Entities

C++ libraries containing references to templated entities can be built using `prelink` and the library generation tools, `ilink` for i-code and `libgen` for object code libraries, respectively. The main thing to keep in mind when creating libraries referencing templates is that all referenced functions and data must be defined within the library itself or within other libraries that are to be used in conjunction with this library. In terms of template instantiation, this translates to saying that all template entities referenced within the library must be present in the library itself as instantiations.

Example

Suppose `stack` is a class template which implements stacks of objects. The source to class template `stack` is contained in the file `stack.h`:

```
template<class T> class stack {
    T *top;
    stack(int size) : top(new T[size]) { ; } // inline

    void push(T val);
    T pop();
};

template<class T>
void stack<T>::push(T val) { *top++ = val; }

template<class T>
T stack<T>::pop() { return *--top; }
```

Further, suppose that a reverse polish notation interpreter using a stack of double values is to be made available in the form of a library `interp.l`. It is natural to use the available class template library `stack`. Then, a function `add()` of this library could be implemented, among other functions, in file `interp.cpp` as:

```
#include "stack.h"
stack<double> stk(100); // stack of double values

double add()
{
    stk.push(stk.pop() + stk.pop());
}
// other functions ...
```

It is important to stress here that we are only interested in building a library which references templated entities as distinguished from building a template library, which is discussed in the next section.

`interp.l` is built by the following steps:

```
xcc -eas interp.cpp
libgen -co=interp.l interp.r
```

Suppose `calc.cpp` is a program which calls functions in `interp.l`, and we try to compile and link it as

```
xcc calc.cpp -l=interp.l
```

the linker reports errors about unresolved symbols

```
stack<T1>::push(T1) [with T1=double]
stack<T1>::pop() [with T1=double]
```

The reason for this is clear from the fact that when the library `interp.l` was built, the member functions `stack<double>::push` and `stack<double>::pop` were not instantiated. Only **requests** for instantiation were generated by the compiler. The user of `interp.l` has no way of instantiating these functions, having no access to the source code for `stack.h`. Thus, to build `interp.l` correctly, we need to make sure that both `stack<double>::push` and `stack<double>::pop` were present as instantiations in `interp.l`. This is done using `prelink` in

addition to `libgen` as:

```
xcc -eas interp.cpp
prelink interp.r
libgen -co=interp.l interp.r
```

The `prelink` step should cause the messages:

```
stack<T1>::push assigned to interp.r [with T1=double]
stack<T1>::pop assigned to interp.r [with T1=double]
executing xcc -eas -fd=interp.r interp.cpp
```

to appear. The file `interp.cpp` is recompiled, but this time the requested instantiations are generated.

As a side note, the linker does not complain about `stack<double>::stack` even if `prelink` is not used when building the library because it is an inline member and these are instantiated whenever referenced.

If `interp.l` is composed of more than one `.r` file, all those that use template functions or classes need to be prelinked together. It is safe to prelink `.r` files that do not reference template functions or classes; these are simply ignored.

To further illustrate the process, suppose that `interp.l` has a function `print()` which is defined in file `print.cpp` as:

```
#include <iostream>
#include "stack.h"

extern stack<double> stk;

void print()
{
    std::cout << stk.pop() << std::endl;
}
```

Again, we build the `interp.l` as:

```
xcc -eas interp.cpp
xcc -eas print.cpp
prelink interp.r print.r
libgen -co=interp.l interp.r print.r
```

Now, suppose that `calc.cpp` also uses `<iostream>`, for instance:

```
// calc.cpp
#include <iostream>
extern void add(), print();
// ...

    add();
    print();
    std::cout << "bye" << std::endl;
```

When we attempt to compile and link `calc.cpp`, we get linker errors again, only this time these are **multiple definition** errors as opposed to the missing definition errors we encountered earlier. The multiple definitions are for templated entities from the `iostream` hierarchy of classes. The reason is that the `prelink` execution causes instantiation of `iostream` template entities in `print.r`. This is because their instantiations are needed in `print.r` and since no other file on the `prelink` command line had the required instantiations, these get instantiated in `print.r`. The problem is that these instantiations (or at least some) are already present in the standard library `cplib.l` and result in multiple definition errors. Thus, `interp.l` should have been built as:

```
xcc -eas interp.cpp
xcc -eas print.cpp
prelink lib-dir/cplib.l interp.r print.r
libgen -co=interp.l interp.r print.r
```

where "lib-dir" denotes the directory where `cplib.l` for the particular target resides.

Building C++ Libraries Referencing Template Entities

1. Compile all C++ and C files comprising the library to i-code or object-code depending on the kind of library to be built.
2. If the library uses templates entities which **might** be present in some other library and which is also going to be linked with the program, then prelink all i-code/o-code files together with that library, else prelink just the i-code/o-code files **even** if there is only one present. Use the `-filink` option if prelinking i-code.
3. As a general rule, have any library that is going to be linked with the program present on the `prelink` command line. Make sure the library path names are correct.

4. Use `libgen` or `ilink` to generate the library

The above procedure is conservative in that it makes no assumptions about what entities could have been instantiated where. One can skip the prelink process if and only if all entities, whose instantiation is requested in any of the i-code/o-code files making up the library, are guaranteed to be defined elsewhere. `libgen -ln` or `idump -n` can be used on an object file/library or i-code file/library respectively and the output piped through `decode` to find out the template instantiation requests. A C++ library should have instantiation requests only for those entities which are present in other libraries (such as `cplib.l`); otherwise there should be none.

Structuring and Building Template Libraries

A C++ template library is a collection of classes and functions in the form of templates contained in header files. Although C++ does not mandate that templates be present in source form, particularly when the new `export` keyword (currently not implemented in Ultra C++) is being used, Ultra C++'s implementation method does require the compiler to have access to template sources when generating instantiations. The easiest way to do this is in the form of header files. The disadvantage with this method is that using such template header libraries means compiling large parts of the library each time the (client) application is compiled. Thus, the benefits of separate compilation are lost. The advantage, however, is the flexibility and type-safety afforded by using templates.

A C++ template library is, in general, composed of two parts: the interface part and the implementation part. The interface is in the form of a header file. Because of the nature of C++ templates, most of the implementation is also present in the header file, but a part of it can be compiled and put into an object-code or i-code library. The following guidelines are recommended for creation of C++ template libraries. Assume that the library implements a class template "container":

- Place the interface for the class template in a header file `container.h`. This means just declaring, and not defining, all member functions of the class template and any other non-member function templates (for instance, friend functions or utility functions).
- Place the definitions of the member functions and the non-member function templates in a header file `container.cc` in the same directory where `container.h` resides. The compiler automatically locates this file when it needs the definitions for instantiation purposes.
- Place the part of the template library that can be compiled in one or more `.cpp` files. For example, the class `container` could use a non-template utility function which can be placed in a file `container.cpp`.
- Any specializations of the class template and any associated function templates must be declared in the header file `container.h`. Since these can be compiled right away, their definitions should be placed in a file which forms part of the compiled library, such as, `container_sp.cpp`.
- Any explicit instantiations can also be compiled right away and should also be placed in a file which forms part of the library, e.g., `container_ei.cpp`.
- If `inline` template functions, whether member or non-member, have to be specialized or explicitly instantiated, this **must** be done in the header `container.h`, that is, their definitions must be present in the header itself rather than just the declarations. This is a requirement of the C++ language.
- Compile all the `.cpp` files, prelink the resulting i-code/object-code files and create a library, say `container.l` or `container.il`, from these as explained in the previous section. Then whenever `container.h` is included, `container.l` or `container.il` needs to be linked in to provide the definitions for any utility functions, specializations, explicit instantiations and any private functions or data which cannot be generated from the templates in `container.h`.

Chapter 9: Object Code Linker

This chapter covers the following linker topics:

- **Purpose**
- **Usage**
- **Execution**
- **Text Output**
- **Library Files**
- **Library Format Created by libgen**
- **Linker Defined Symbols**
- **Module Header Override Symbols**
- **Linking Code for Non-OS-9 Systems**



Purpose

The linker transforms the ROF produced by the assembler into a single OS-9 format memory module. A memory module must minimally consist of the following:

- A module header
- A module body
- A cyclic redundancy check (CRC) value

Many modules require more than this basic information. For example:

- Program and trap handler modules require data memory and stack memory.
- File manager, device driver, and device descriptor modules all require special information unique to each type of module.

The contents of the assembly language ROFs provide the linker with the information required to create each type of memory module.



For More Information

Your operating system technical manual contains additional information about memory modules.

psects

A program usually consists of many small code segments which, when processed by the linker, form the final executable memory module. Each code segment is called a `psect`. The `psect` is the basic unit on which the linker operates. The `psects` are stored in the ROFs produced by the assembler. The `psect` provides the linker with the following information:

- Identifying information about the `psect`
- Size of the code, data, initialized data, and remote memory area
- Symbols defined by the individual `psect`
- Symbols referenced by the `psect`
- Relocation information
- Actual code and initialized data

Usage

Because the name of the linker may differ from processor to processor, this manual refers to the resident linker as **linker**.



For More Information

For valid linker names, refer to [Chapter 6: Assembler and Object Code Linker Overview](#).

The linker command line has the following syntax:

```
linker [options] <mainline> [<rof2> {<rofN>} ]
```

`options` specifies any of the options listed in the next section.



For More Information

Refer to [Chapter 5: Compiler Phase Options](#), for a list and descriptions of object code linker options.

`mainline` is the pathlist of the file containing the root `psect` from a module header is generated. A non-zero `type` or `lang` value in the `psect` directive indicates a root `psect`.

`rof2` through `rofN` specify the names of additional ROFs. `rof2` through `rofN` cannot contain a root `psect`. The root and all subroutine files appear in the final linked object module regardless whether they were actually referenced. The number of ROFs used is not limited. All linker input files must be in either ROF or library format (a library created using the `libgen` utility). Multiple library formats are available. The format used is based upon the processor.



For More Information

Refer to the appropriate processor chapter in the *Ultra C/C++ Processor Guide* to determine the library format used by a processor.



Note

Processors using ROF edition number 9 use library format number 1.
Processors using ROF edition number 15 use library format number 3.

Text Output

The linker `-m` option produces text output for each `psect` it processes. The output is formatted as follows.

```
'<psect_name>'psect from file: <filename>
C:<hex val> M:<hex val> D:<hex val> RM:<hex val>
      RD:<hex val>
```

where:

C: is the beginning offset of the `psect` code area

M: is the beginning offset of the `psect` uninitialized static memory

D: is the beginning offset of the `psect` initialized static memory

RM: is the beginning offset of the `psect` remote un-init statics

RD: is the beginning offset of the `psect` remote init statics

The following is an example of the output produced with the `-m` option.

```
'memory.c'psect from file: /dd/MWOS/OS9/68000/LIB/
      clib.1
C:00002ba2 M:00000d5a D:00000e5c RM:000011d6
      RD:000011d6
```

The linker `-s` option produces text output for each `psect`. The output format is similar to the following.

```
<symbol 1> <type> <hex val> | <symbol 2> <type>
      <hex val> | <symbol 3> <type> <hex val>
```

where `<type>` is one of the symbol types identified in [Table 9-1](#).

Table 9-1 Symbols Types

<type>	Description
COD	Code
MEM	Uninitialized static memory

Table 9-1 Symbols Types (continued)

<type>	Description
DAT	Initialized static memory
RME	Remote uninitialized static memory
RDA	Remote initialized static memory
ABS	Absolute value (not a relative offset)

The following is an example of the output produced with the `-s` option

```

_lmalloc COD 00002c2a | malloc COD 00002dba | _lrealloc COD 00002dec |
_lfree COD 00002f72 | realloc COD 0000304a | free COD 000030ae |
_freemin COD 000030dc | _mallocmin COD 00003134 | calloc COD 0000317e |
_lcalloc COD 000031c0 | memptr MEM 00000d5a | membegin COD 0000e60 |

```

Execution

During program assembly, the assembler does not recognize the addresses of names that are external references to other program sections. For example, the Motorola 68K `bsr` or the Intel 386 `call` instruction to a label in another program section cannot have its offset computed because the address of the destination label is not known until the linker combines all sections. Therefore, when an external reference is encountered, the assembler sets up information in the ROF that identifies the instructions that reference external names. Because the assembler is not aware of what the actual offset within the module will be, each section is assembled as though it starts at offset 0.

The linker uses the ROFs produced by the assembler as input. The linker reads all the ROFs and then assigns each ROF a relative starting offset for its data storage space and a relative starting offset for its object code space.

Some processors support addressing modes that allow data to be accessed by using both positive and negative offsets from some base address. For example, the Motorola 68K processor family supports an address register indirect mode which allows access to data at any offset from -32768 to 32767 from the base address register.



For More Information

Refer to the Assembler/ Linker section in the appropriate processor chapter in the *Using Ultra C/C++ Processor Guide* for information on biasing.



Note

Because OS-9 requires programs to be position-independent code with separate position-independent data areas, these addresses remain relative. The operating system assigns these physical memory areas when a program is loaded and executed.

The linker processes the input files in two phases as described in the following subsections.

First Phase

During the first phase, the linker reads all the input files in the order they appear on the command line and checks each `psect` for validity. The global symbol definitions are entered into the defined symbol table. If a symbol of the same name already exists in the defined symbol table, an error message is generated. The unresolved references are also gathered.

After all of the ROFs are read, each undefined symbol is checked against the global symbol table. If found, the symbol is removed from the undefined symbol table.

If, after examining the input files, the linker still has unresolved symbols, it reads the library files. The library files are processed one at a time until either no unresolved references exist or the end of the library list is reached. The linker reports any unresolved references at the end of the library search as errors.

The linker handles the symbol `psect` as a special case. A symbol `psect` contains no code or data, only symbols defining constants. When a symbol `psect` is processed, only the symbols marked as undefined are placed in the defined symbol table. This procedure minimizes the amount of symbol table memory required for linking modules against the system library. When using a symbol `psect`, place it last in the list of ROF files specified on the linker command line. This ensures that all references that it should resolve will have been found.

During the first phase, the linker determines the size of the code, data, initialized data, and remote memory areas. The offsets of all code symbols are assigned based on each `psect`'s position in the final module, and the proper symbol bias values are applied.

Second Phase

During the second phase, the linker creates the output module. The module header for the appropriate module type is created and written to the output file. Each input `psect` is re-read from the appropriate input or library file. The code and initialized data segments are read into an internal buffer.

The reference list in the `psect` is read to determine the locations of all operands referencing external symbols. These operands are adjusted to reflect the destination position in the output module.

The code and initialized data segments are then written to the output file. As each segment is written, the module CRC is calculated. The CRC is written into the output module when all `psects` have been processed.

The following example is a simplified memory map showing the memory allocation for three segments (A, B, and C) after processing by the linker.

Figure 9-1 Executable Memory Module

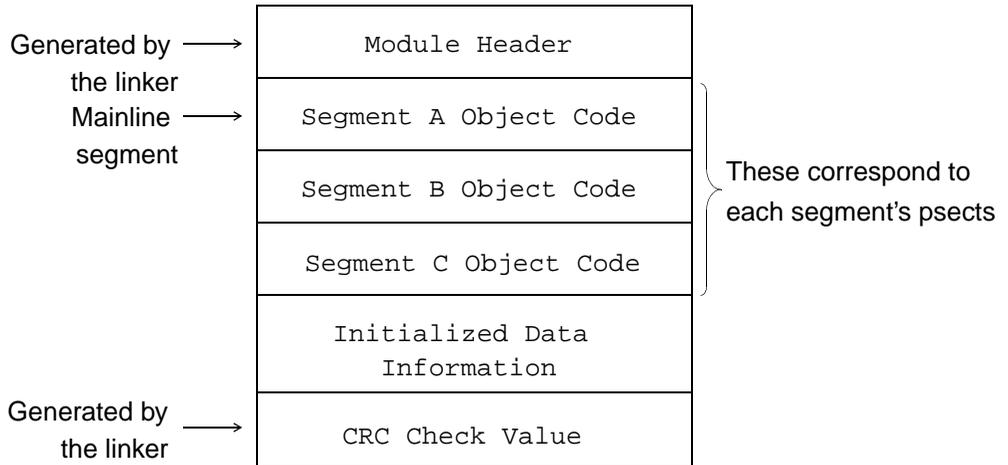
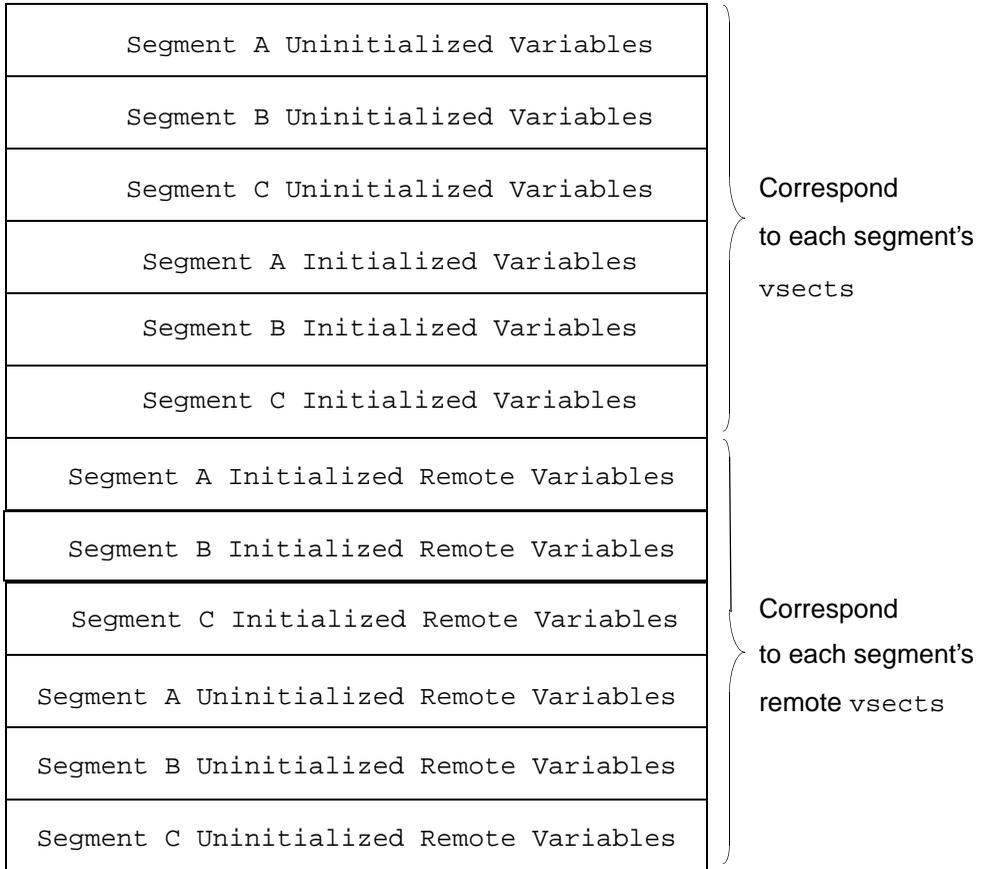


Figure 9-2 Process Data Area



Library Files

The `libgen` and `merge` methods of creating libraries are described in the following sections.

Libgen

The `libgen` utility creates libraries and displays library information. `libgen` processes a group of ROFs into a fast linking format.



For More Information

Chapter 10 contains a description of `libgen`.

In general, `libgen`:

- Resolves references internal to the library, eliminating the ordering issues previously described
- Groups all like ROF sections together
- Makes a list of unresolved references for the entire library

Merged Libraries

A simple library file can also be created by concatenating one or more ROFs into a single file using the `merge` utility. To change a single `psect` in such a library file, the entire library must be re-created from the ROFs, substituting the new `psect` for the old.



For More Information

Refer to the *Utilities Reference* for information on `merge`.

The linker performs only one pass over the input files to locate symbol definitions. Because of this, the order in which the `psects` appear in a simple library file is important. The `psects` must be ordered so that the references are generally forward references. Consider the following example:

```
psect main_c
defines:  main
references:  sub_1

psect sub1_c
defines:  sub_1, sub1a
references:  sub2

psect sub2_c
defines:  sub2
references:  printf
```

The `psect sub1_c` must appear in the library before any `psect` containing a symbol that `sub1_c` references. If the `sub2_c` `psect` were to appear before the `sub1_c` `psect`, the symbol definition for `sub2` would not be found.

Use the `-l` option of `rdump` to examine this library relationship in a merged library.



For More Information

Chapter 10 contains a description of `rdump`.

Library Format Created by libgen

Two library formats are: type 1 and type 3. To determine the appropriate library format for a processor, reference the appropriate chapter of the *Ultra C/C++ Processor Guide*. Processors using ROF edition number 9 use library format number 1. Processors using ROF edition number 15 use library format number 3.

The format for libraries created by `libgen` includes the following sections.

- Library header
- Global definition section
- `psect` section
- Internal reference section
- External reference section
- Object code sections for all ROFs processed into library
- Reference and etree sections for all ROFs processed into the library

Library Header

Table 9-2 identifies the format for the library header.

Table 9-2 Library Header

Format Type 1 (bytes)	Format Type 3 (bytes)	Description
4	4	Library identification code
2	2	Library format type
6	6	Date the library was made

Table 9-2 Library Header (continued)

Format Type 1 (bytes)	Format Type 3 (bytes)	Description
2	2	Library edition number
4	4	Offset in string table of library name
4	4	Size of the global definition hash table (in bytes)
4	4	Size of the library global definition section (in bytes)
4	4	Size of the library string table (in bytes)
4	4	Size of the library <code>psect</code> section (in bytes)
4	4	Size of the library internal reference section (in bytes)
4	4	Size of the library external reference section (in bytes)
	4	Header expansion (reserved for future use)



For More Information

For more information about sections for all ROFs processed into the library, refer to [Chapter 6](#).

Global Definition Hash Table

The global definition hash table is a table of four-byte indices into the global definition section, hashed by global definition name. The number of entries in the hash table is `<hash table size>` divided by four. For example, a hash table with a size of 24 bytes has six entries of four bytes each.

Global Definition Section

Table 9-3 identifies the format for global definition entries.

Table 9-3 Global Definition Entries

Bytes	Description
2	Symbol type (reference Chapter 6 , for symbol types)
4	Symbol value
4	Offset of the symbol name in the string table
4	Index of the next global definition entry in the hash chain
4	Index of the next global definition entry in the list for the <code>psect</code>
4	Index of the <code>psect</code> entry for the <code>psect</code> containing this symbol

String Table

The string table contains `<string table size>` bytes of name strings.

psect Section

The `psect` section contains a table of `psect` information entries. Entries are for library format type 1 in [Table 9-4](#) and library format type 3 in [Table 9-5](#).



Note

A `psect` appearing in a library file is retained for the final module only if the `psect` defines a symbol that was undefined before the library was searched.

Table 9-4 Library Format Type 1 psect Information Entries

Bytes	Description
2	Format type of the ROF that contained the <code>psect</code>
4	Size of the ROF uninitialized static storage
4	Size of the ROF initialized static storage
4	Size of the ROF remote uninitialized static storage
4	Size of the ROF remote initialized static storage
4	Size of the ROF object code

Table 9-4 Library Format Type 1 psect Information Entries (continued)

Bytes	Description
4	Size of the ROF debug information
4	Size of the ROF stack required
4	Offset of the object code in the library
4	Offset of local references in the library
4	Number of references in the ROF to remote data
4	Number of references in the ROF to code
4	Index in the internal reference section to the head of the internal reference list for the ROF
4	Index in the external reference section to the head of the external reference list for the ROF
4	Offset of <code>psect</code> name in the string table
4	Index in the global definition section to the head of the global definition list for the ROF

Table 9-5 Library Format Type 3 psect Information Entries

Bytes	Description
2	Format type of the ROF that contained the <code>psect</code>
4	Size of the ROF uninitialized static storage
4	Size of the ROF initialized static storage
4	Size of the ROF constant static storage
4	Size of the ROF remote uninitialized static storage
4	Size of the ROF remote initialized static storage
4	Size of the ROF remote constant static storage
4	Size of the ROF object code
4	Size of the ROF debug information
4	Size of the ROF stack required
2	Target processor type
4	Expansion (reserved for future use)
4	Offset of the object code in the library
4	Offset of local references in the library
4	Offset of expression tree data in the library
4	Offset of the reference data in the library

Table 9-5 Library Format Type 3 psect Information Entries (continued)

Bytes	Description
4	Number of references in the ROF to remote data
4	Number of references in the ROF to code
4	Index in the internal reference section to the head of the internal reference list for the ROF
4	Index in the external reference section to the head of the external reference list for the ROF
4	Offset of <code>psect</code> name in the string table
4	Index in the global definition section to the head of the global definition list for the ROF

Internal Reference Section

The internal reference section contains the table of internal reference entries. These references are resolved within the library.

Table 9-6 Internal Reference Section

Bytes	Description
2	Number of references to the symbol
4	Offset in the library of reference information for the symbol

Table 9-6 Internal Reference Section (continued)

Bytes	Description
4	Index of the next internal reference entry in the list for the <code>psect</code>
4	Index in the global definition section of global definition that resolves references

External Reference Section

The external reference section contains a table of symbol references external to the library. These references are resolved outside of the library or reported as unresolved references at link time.

Table 9-7 Symbol References External to the Library

Bytes	Description
2	Number of references to the symbol
4	Offset in the library of reference information for the symbol
4	Index of the next external reference entry in the list for the <code>psect</code>
4	Index in the string table of the symbol name referenced

Linker Defined Symbols

The linker defines some symbols at link time to enable access to values that cannot be determined until the final code and data offsets are known.



Note

Each offset listed will be biased by the linker as appropriate for the platform and module type being targeted.

Without the `-c` option, the linker generates the code and data symbols identified in [Table 9-8](#) and [Table 9-9](#).

Table 9-8 Code Symbols without -c Option

Code Symbols	Description
<code>bname</code>	Offset from the beginning of the module to the module name
<code>_bname</code>	Same as <code>bname</code> . Offset from the beginning of the module to the module name. (ANSI/ISO C compliant.)
<code>btext</code>	The symbol used to refer to the beginning of the module. As this is calculated as a biased offset from the beginning of the module, <code>btext</code> is zero biased by the appropriate value.
<code>_btext</code>	The symbol used to refer to the beginning of the module. As this is calculated as a biased offset from the beginning of the module, <code>btext</code> is zero biased by the appropriate value.

Table 9-8 Code Symbols without -c Option (continued)

Code Symbols	Description
<code>etext</code>	Offset from the beginning of the module to the end of the module
<code>__etext</code>	Same as <code>etext</code> . Offset from the beginning of the module to the end of the module. (ANSI/ISO C compliant)

Table 9-9 Data Symbols without -c Option

Data Symbols	Description
<code>__jumptbl</code>	Offset to the jumtable
<code>end</code>	Last data offset assigned
<code>__enddata</code>	Same as <code>end</code> . Last data offset assigned (ANSI/ISO C compliant)

In addition to the symbols identified in [Table 9-8](#) and [Table 9-9](#), if the `-r` option is on without the `-c` option, the `etext` description changes and additional symbols are generated as described in [Table 9-10](#).

Table 9-10 Code Symbols with `-r` Option, without `-c` Option

Code Symbols	Description
<code>etext</code>	Offset to the beginning of the initialized data
<code>_bidata</code>	Same as <code>etext</code> . Offset to the beginning of the initialized data (ANSI/ISO C compliant).
<code>edata</code>	Offset to the beginning of the <code>irefs</code> section (the end of the initialized data section)
<code>_birefs</code>	Same as <code>edata</code> . Offset to the beginning of the <code>irefs</code> section (the end of the initialized data section) (ANSI/ISO C compliant)

If the `-c` option is on, the linker generates only the code and data symbols identified in [Table 9-11](#) and [Table 9-12](#).

Table 9-11 Code Symbols with `-c` Option

Code Symbols	Description
<code>_bname</code>	Offset from the beginning of the module to the module name (ANSI/ISO C compliant)
<code>_btext</code>	Offset from the beginning of the module to the beginning of the module (ANSI/ISO C compliant)
<code>_etext</code>	Offset from the beginning of the module to the end of module (ANSI/ISO C compliant)

Table 9-12 Data Symbols with -c Option

Data Symbols	Description
<code>__jumptbl</code>	Offset to the jumtable. (ANSI/ISO C compliant)
<code>__enddata</code>	Last data offset assigned. (ANSI/ISO C compliant)

In addition to the symbols identified in [Table 9-11](#) and [Table 9-12](#), if `-r` option is on and the `-c` option is on, the linker generates the symbols described in [Table 9-13](#).

Table 9-13 Code Symbols with Both -r and -c Options

Code Symbols	Description
<code>__etext</code>	Offset to the end of the code section. (ANSI/ISO C compliant)
<code>__bidata</code>	Offset to the beginning of the initialized data section. (ANSI/ISO C compliant)
<code>__birefs</code>	Offset to the beginning of the <code>irefs</code> section. (ANSI/ISO C compliant)

If `-r` is on, additional absolute symbols are defined to provide link-time constants to the user. These are identified in [Table 9-14](#).

Table 9-14 Absolute Symbols with `-r` Option without `-c` Option

Absolute Symbols	Description
<code>dsize</code>	Size of the accumulated data area from all linked <code>psects</code> , rounded up to a 256 byte boundary.
<code>_dsize</code>	Same as <code>dsize</code> . Size of the data area rounded up to a 256 byte boundary. (ANSI/ISO C compliant)
<code>_codebias</code>	The bias added by the linker to code symbols. (ANSI/ISO C compliant)
<code>_databias</code>	The bias added by the linker to data symbols. (ANSI/ISO C compliant)

If the `-c` option is on with the `-r` option, the linker generates the absolute symbols identified in [Table 9-15](#).

Table 9-15 Absolute Symbols with Both `-r` and `-c` Options

Absolute Symbols	Description
<code>_dsize</code>	Size of the data area rounded up to a 256 byte boundary (ANSI/ISO C compliant).
<code>_codebias</code>	The bias added by the linker to code symbols. (ANSI/ISO C compliant)
<code>_databias</code>	The bias added by the linker to data symbols (ANSI/ISO C compliant)

Module Header Override Symbols

The linker recognizes certain global labels as overrides to selected fields in the module header. The linker places the value of these symbols into the appropriate field of the module header.

Table 9-16 Global Label Overrides

Symbols	Description
<code>_m_access</code>	Permissions
<code>_m_attrev</code>	Attribute/revision value
<code>_m_edit</code>	Edition number (supersedes <code>_sysedit</code> global label)
<code>_m_grpusr</code>	Module owner
<code>_m_init</code>	Init routine entry point (OS-9 only, not 68K)
<code>_m_term</code>	Term routine entry point (OS-9 only, not 68K)
<code>_m_share</code>	Shared memory offset (OS-9 only, not 68K)
<code>_m_usage</code>	Comment string (OS-9 for 68K only)
<code>_m_tylan</code>	Type/Language value
<code>_m_exec</code>	Execution entry point
<code>_m_excpt</code>	Exception entry point
<code>_m_stack</code>	Stack requirement
<code>_sysattr</code>	Attribute/revision value
<code>_sysedit</code>	Edition number (superseded by <code>_m_edit</code> global label)

Table 9-16 Global Label Overrides (continued)

Symbols	Description
<code>_sysperm</code>	Permissions
<code>_syscmnt</code>	Comment (OS-9 for 68K only)

Override symbols are typically set with the `equ` directive as:

```
_m_edit: equ 21;edition number
_m_access: equ PRead_|Read;module access permissions
_m_attrev: equ (ReEnt|Ghost)<<8|revision
* ;module attributes
```

Linking Code for Non-OS-9 Systems

The linker can generate raw code to run in non-OS-9 environments. The output is a pure binary file that is not in OS-9 memory module format.

Use the linker `-r` option to create raw output files. The hexadecimal address to place the modules in ROM is specified using the `-r` option. The address enables resolution of absolute references.

The initialization code must set up the stack pointer to point to a stack RAM area. The appropriate register must also point to the beginning of a global/static RAM area (`vsect`) that you should initialize to zeros. Finally, the initialized data information must be processed.

Some processors may require biasing and the initialization of a code area data pointer with the proper bias value applied.



For More Information

Refer to the C ABI and Assembler/Linker sections in the appropriate processor chapter in the *Using Ultra C /C++ Processor Guide* for information concerning stack pointer, static storage pointer, and data biasing for your processor.

Chapter 10: Utilities

The following utilities are available with Ultra C/C++:

- **deasm**
- **decode**
- **idump**
- **libgen**
- **rdump**



deasm

Convert Microware K&R Style Assembly Language to Ultra C/C++

Syntax

```
deasm [<opts>] {<file>}
```

Description

deasm converts the Microware K&R compiler style assembly language in a source file to Ultra C/C++ assembly language style.

deasm reads standard input if no file name is given on the command line. It writes to standard output, unless the `-o` option is used to specify an output file name.

Options

The following options are available with deasm:

<code>-p</code>	Indent at (@) lines
<code>-o[=]<file></code>	Specify the name of the output file

Example

```
deasm <in.c >out.c
deasm in.c -o=out.c
```

Convert `in.c` to Ultra C/C++ assembly language style in file `out.c`.

For the input file (`in.c`):

```
/* sign extend a word */
#asm
sign_word: ext.l d0 sign extend d0
          rts
#endasm
/* sign extend a byte */
@sign_byte: ext.w d0 sign extend d0
@ ext.l d0
@ rts
```

The command line `deasm in.c -p -o=out.c` generates:

```
/* sign extend a word */
_asm("
sign_word: ext.l d0 sign extend d0
      rts
");
/* sign extend a byte */
_asm("sign_byte: ext.w d0 sign extend d0 ");
_asm("      ext.l d0 ");
_asm("      rts ");
```

decodeName Demangler

Syntax

```
decode [ <opt> ]
```

Description

`decode` is a C++ name demangler. Things that look like mangled names are demangled. `decode` reads input from `stdin`, and writes to `stdout`. Everything else is passed through unchanged.

Options

<code>-u</code>	Ignore extra underscore in mangled names
-----------------	--

idump**Dump Symbol Information from I-Code Files**

Syntax

```
idump [<opts>] {<I-code file>}
```

Description

`idump` prints the symbol-related information from an I-code file or library. The information may be used determine the symbols (with their type information) that an I-code file references and defines.

`idump` displays a header for each I-code section. The header contains:

ISect name	Name of the source file used to generate the I-code section. This is only used for diagnostic identification.
I-code Rev	Revision of the I-code format for that I-code section. The compiler uses this to ensure that a particular executable can read the I-code section.
Host and Target	The compiler uses this to distinguish between different hosts and targets.
Debug	Displays Yes if the I-code section has source level debugging information associated with it. Otherwise, No is displayed.
Valid	<p>The compiler uses the following flags to determine the phases that have finished with the I-code file.</p> <p>COMP The front end has successfully compiled the source into the I-code file.</p> <p>OPT1 The I-code optimizer has made a pass over the file.</p>

threads	Determines whether the code contained has no support (<code>none</code>) for threads; is thread using, <code>safe</code> ; or <code>thread safe</code> .
Date	The date when the front end generated the I-code section.

Options

`idump` options determine the amount of additional information printed:

-a	All code and data symbols defined and referenced with type information
-e	External code and data symbols referenced
-ec	External code symbols referenced
-ed	External data symbols referenced
-g	Global code and data symbols defined
-gc	Global code symbols defined
-gd	Global data symbols defined
-t	Type information for all symbols displayed

Example

The following is example output of `idump` on a single I-code file:

```
ISect name: prog.c
I-code Rev: 30
  Host: 0
  Target: 0
  Debug: No
  Valid: COMP OPT1
Threads: none
  Date: Fri May 7 09:42:14 1993
Global Information
  Code Symbols
  main - func ret void
```

```

Data Symbols
glob - 32-bit
Str - ptr to composite(size 18, align 2)
str - composite(size 18, align 2)
External Reference Information
Code Symbols
__ansi_printf - func ret 32-bit
exit - func ret 32-bit
Data Symbols
errno - 32-bit

```

`prog.c` was compiled by a compiler that generates revision 30 I-code format for host 0 and target 0, without source level debugging information, and with I-code optimization enabled on Friday May 7, 1993.

`prog.c` defined one code symbol (`main`) as a function returning void. It defined three data symbols:

<code>glob</code>	A 32-bit integer
<code>Str</code>	A pointer to an array, structure, or union of size 18 bytes and even alignment
<code>str</code>	An array, structure, or union of size 18 bytes and even alignment

`prog.c` references two external functions, `__ansi_printf` and `exit`, that return 32-bit integer values. It also refers to an external global, `errno`, that is a 32-bit integer.

libgen

Create Libraries and Display Library Information

Syntax

```
libgen [<opts>] [<files>]
```

Description

`libgen` processes a group of ROFs into a fast linking format. In general, `libgen`:

- Resolves references internal to the library
- Groups all like ROF sections together
- Makes a list of unresolved references for the entire library

`files` is the group of ROFs to process.

Options

The following options are available with `libgen`:

<code>-b=<n></code>	Use an output buffer size of <code><n></code> kilobytes
<code>-c</code>	Create a library
<code>-e=<number></code>	Set the library edition number
<code>-f=<path></code>	Output file for '-l' output
<code>-l</code>	List the names of the modules in the library
<code>-le</code>	List extended information about the modules in the library
<code>-li</code>	List the identification information for the library
<code>-ll</code>	List the local references for modules in the library
<code>-ln</code>	Format displayed information is a style similar to that used by the UNIX <code>nm</code> utility

<code>-lu</code>	List the names of modules in the library, one file per line (unformatted)
<code>-mts</code>	Mark a library as inherently thread safe. Used to make non-thread using libraries compatible with thread using libraries. Must be used with the <code>-c</code> option.
<code>-o=<path></code>	Output file for library to <code><path></code>
<code>-p=<psect></code>	Only list information from specified <code><psect></code>
<code>-z [=<path>]</code>	Read file names from standard input or file (<code><path></code>)

rdump

Examine Contents of Library Files

Syntax

```
rdump {<rof>} [<opts>]
```

Description

`rdump` can be used to examine the content of ROFs or simple library files.

`<rof>` must be a relocatable object file or simple library. It usually has a suffix of `.r` or `.l`.

Options

<code>-a</code>	Display all information generated by the <code>-g</code> , <code>-r</code> , <code>-o</code> , <code>-e</code> , and <code>-c</code> options
<code>-c</code>	Display the code text file offset
<code>-e</code>	Display the expression tree information (ROF Edition Number 14 and above only)
<code>-f=<filename></code>	Send output to specified file
<code>-g</code>	Display the global definition information
<code>-l</code>	Merged ROF library ordering check
<code>-o</code>	Display the reference and local offset information
<code>-r</code>	Display the external reference information

Chapter 11: Optimizations

Ultra C/C++ performs numerous optimizations to improve execution speed and reduce the size of the final object code. Optimization algorithms from the latest conference proceedings, academic research, and industry research are incorporated. Ultra C/C++ performs the following optimizations:

- **Constant Propagation**
- **Constant Folding**
- **Loop Rotation**
- **Variable Lifetimes**
- **Register Coloring and Coalescing**
- **Common Subexpressions**
- **Pointer Tracking with CSE**
- **Useless Code Elimination**
- **Useless Copy Elimination**
- **Useless Pointer Elimination**
- **Assignment Translation**
- **Code Motion and Combining**
- **Loop Optimizations**
- **Constant Sharing**
- **Function Inlining**
- **Span Dependent Optimizations**
- **Assembly Level Optimizations**



Constant Propagation

If the compiler determines that a variable contains a constant value, it replaces references to this variable with the constant resulting in opportunities to fold the expression.

Constant Folding

The compiler computes expressions involving only constants at compile time. This is often the case when the `sizeof()` operator or preprocessor macros are used.

For example:

```
struct a *ptr;  
ptr = grab(100 * sizeof(struct a));
```

The expression ‘100 objects the size of structure *a*’ must be used because the size of structure *a* is unknown to the programmer. At compile time, the size of structure *a* is known. For example, if the size of structure *a* is known to be eight at compile time, the compiler can generate:

```
ptr = grab(800);
```

Because multiplication is not performed at run time, the executable module is smaller and faster.

Loop Rotation

The `for` and `while` loops are the standard loop constructs in C. Both loops require a test of the exit condition before entering the loop. The compiler places the code to test the exit condition at the bottom of the loop and a branch before the loop that goes over the body of the loop to the test (a branch that is executed only once). This reduces the number of branches executed by the number of times through the loop minus one.

Variable Lifetimes

The compiler computes a **weighted lifetime** for each local variable.

- **Weight** specifies the number of times that the variable is likely to be referenced over a range of code. The compiler uses this number when deciding which variables, if any, to put on the stack.
- **Lifetime** specifies the ranges of code for which a specific value of a variable is needed. The lifetime information is used for register coloring and coalescing.

For example, in the following function, the compiler recognizes that `i` and `j` are not used simultaneously. With this knowledge, the compiler may allocate the same register to both variables providing optimal register use, decreased code size, and a faster executable:

```
main()
{
    int i, j;

    for (i = 0; i < 100; i++)
        func(i);
    for (j = 0; j < 100; j++)
        func(j);
}
```

Register Coloring and Coalescing

The compiler performs both register coloring and register coalescing.

- Register coloring determines the best use of registers for a function and puts the least used variables on the stack.
- Register coalescing reduces register-to-register moves by determining the most efficient placement of variables and compiler temporaries.

Register coalescing prevents unnecessary data movement. For example, consider the following code:

```
a = b + 10
```

A compiler that does not perform register coalescing could generate the following code:

```
move.l d4,d0* load b into a temporary register
add.l #10,d0* add 10 to the temporary
move.l d0,d5* store the temporary in a different
               * register
```

Because Ultra C/C++ performs register coalescing, it can generate the following code:

```
move.l d4,d5* load b into a
add.l #10,d5* add 10 to it
```

And, if this is the last reference to `b`, the compiler can perform further coalescing:

```
add.l #10,d4* add 10 to the old value in b to become a
```

Common Subexpressions

The compiler recognizes when an expression, or portion of an expression, is used multiple times. If this expression is expensive to compute, the compiler places the common expression in a temporary variable. Instead of recomputing the expression, future occurrences of the expression reference the temporary variable. For example:

```
a = b * c + 5;  
func(b * c);
```

This could be changed to the following to save a recomputation of `b * c`:

```
a = (t = b * c) + 5;  
func(t);
```

This optimization is effective even on code not containing obvious common subexpressions. Often, the underlying generated code for array and structure members contains common subexpressions that may be eliminated. On some processor architectures, it may be beneficial to consider certain constants, addresses of functions and addresses of global variables for common subexpression elimination. This is because recomputation of these values can be expensive on these architectures. This optimization reduces code size and improves execution speed.

Pointer Tracking with CSE

The compiler tracks the assignment and use of pointer type variables in a function. This enables the compiler to know which values are destroyed when a value is stored at a pointer's destination. Consider the following code sequence:

```
int a, b, c, d, *p;

1  p = &d;
2  a = b * c + 5;
3  *p = 10;
4  func(b * c);
```

The compiler assigns the $b * c$ in line 2 into a temporary variable and refers to it in line 4 instead of $b * c$ again. Before doing this, the compiler must ensure that line 3 does not change the value of b or c . Because the compiler has previously defined p as a pointer to d , the compiler knows that $*p = 10$ does not alter b or c . Therefore, the code may be changed to the following:

```
1  p = &d;
2  a = (t = b * c) + 5;
3  *p = 10;
4  func(t);
```

Useless Code Elimination

The compiler examines code to find unnecessary assignments into variables. These assignments are eliminated along with the expression being placed in the variable (barring any side-effects) and often leads to further eliminations. This optimization reduces code size and improves execution speed by removing ineffective code.

For example, consider the function:

```
f()
{
    int a, b;
1   a = func(1);
2   b = a * 10;
3   return func(2);
}
```

Because `b` is not used after it is given a value, line 2 may be removed, leaving the following:

```
f()
{
    int a;
1   a = func(1);
2   return func(2);
}
```

The compiler may remove the assignment to `a` in line 1 because the reference to `a` was removed. However, the call to `func()` must remain because removing it may cause a side effect. This leaves the following:

```
f()
{
1   func(1);
2   return func(2);
}
```

Eliminating useless code also streamlines the code generated by the front end. The code generated by the front end may not be optimal due to the one-pass nature.

Streamline optimizations eliminate the following:

- one branch to another branch
- switch cases that refer to the default case

Useless Copy Elimination

Useless copy elimination enables the compiler to eliminate variables that exist only to hold the value of another variable. Useless copies are often generated after inlining a function into another function. This optimization reduces code size and improves execution speed. For example:

```
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
int func(int a, int b)
{
    int m;
    m = max(a, b);
    return (m / 2) + a;
}
```

After inlining, could be changed to:

```
int func(int a, int b)
{
    int m, x, y;
    x = a;
    y = b;
    if (x > y)
        m = x;
    else
        m = y;
    return (m / 2) + a;
}
```

Note the useless copies of a in x and b in y . With useless copy elimination the function is changed to:

```
int func(int a, int b)
{
    int m;
    if (a > b)
        m = a;
    else
        m = b;
    return (m / 2) + a;
}
```

Useless Pointer Elimination

Useless pointer elimination enables the compiler to eliminate pointers when actual objects may be used in their place. This occurs most frequently after inlining a function into another function. Since C disallows the return of more than one value, the alternative is to pass the address of other returned values as parameters. This optimization reduces code size and improves execution speed. For example:

```
int glob;
int ret(int code, int *status)
{
    if (code == 2)
        *status = errno;
    else
        *status = 0;
    return code;
}
func()
{
    int i, s;
    i = ret(glob, &s);
    printf("i = %d, s = %d\n", i, s);
}
```

After inlining, could be changed to:

```
int glob;
func()
{
    int i, s;
    int code, *status;
    code = glob;
    status = &s;
    if (code == 2)
        *status = errno;
    else
        *status = 0;
    i = code;
    printf("i = %d, s = %d\n", i, s);
}
```

Useless pointer elimination changes the function to:

```
func()
{
    int i, s;
    int code;
    code = glob;
    if (code == 2)
        s = errno;
    else
        s = 0;
    i = code;
    printf("i = %d, s = %d\n", i, s);
}
```

Assignment Translation

The compiler scans the code for assignments that it can turn into assignment operators. For example:

```
i = x / 10 + func(b) + i;
```

This can be translated to:

```
i += x / 10 + func(b);
```

Assignment translation reduces code size and improves execution speed.

Code Motion and Combining

The compiler looks for common sequences of code and moves them. This movement occurs under three different circumstances: common successor code, common predecessor code, and common tail code.

Common Successor Code

Common successor code occurs when the code that starts each path from a branch is identical. For example:

```
if (a) {  
    b = c * 10;  
    func(a);  
}  
else {  
    b = c * 10;  
    func(b);  
}
```

Because `b = c * 10` is executed regardless of the value of `a`, it can be moved before the test. This saves a copy of an identical code section:

```
b = c * 10;  
if (a) {  
    func(a);  
}  
else {  
    func(b);  
}
```

Common Predecessor Code

Common predecessor code occurs when branches are merging into a common point. If the code at the end of all merging branches is the same, it can be dropped from each branch. For example:

```
if (a) {
    func(a);
    b = c * 10;
}
else {
    func(b);
    b = c * 10;
}
```

This can be changed to:

```
if (a) {
    func(a);
}
else {
    func(b);
}
b = c * 10;
```

Common Tail Code

Common tail code occurs when a function has several returns and some have common code related to the return. The compiler can merge this common code to form a single tail. For example:

```
b = c * 10;
return b;
.
.
.
b = c * 10;
return b;
```

This can be changed to:

```
goto label1;  
.  
.  
.  
label1:  
    b = c * 10;  
    return b;
```

These optimizations make the executable module smaller, but have little effect on execution speed.

Loop Optimizations

The compiler performs four loop optimizations:

- **Initial Loop Condition Testing**
- **Invariant Hoisting**
- **Strength Reduction**
- **Loop Unrolling**

Initial Loop Condition Testing

Initial loop condition testing occurs when the compiler determines that the first test of a loop condition is true. Both `for` and `while` loops in C test the exit condition before executing the body of the loop. If the compiler determines that the first test of the loop condition is true, it changes the loop so that the exit condition is not tested until the loop body has executed once. For example:

```
func()
{
    int i;
    i = 0;
    while (i < 1000) {
        array[i++] = rand();
    }
}
```

This could be changed to:

```
func()
{
    int i;
    i = 0;
    do {
        array[i++] = rand();
    } while (i < 1000);
}
```

This results in a slightly improved execution speed and smaller code size.

Invariant Hoisting

Invariant hoisting occurs when code placed within a loop does not change. The compiler moves this code outside the loop to prevent needless computations. Although this optimization increases code size, execution speed is improved. For example:

```
for (i = 0; i < 100; i++)  
    func(a * b);
```

Because neither `a` nor `b` change as the loop iterates, the loop may be changed to:

```
t = a * b;  
for (i = 0; i < 100; i++)  
    func(t);
```

This case may be performed manually but when the invariant information is implicit it is less obvious. Consider the loop:

```
for (x = 0; x < 100; x++)  
    for (y = 0; y < 100; y++)  
        array[x][y] = func(x, y);
```

The compiler could change the loop to:

```
for (x = 0; x < 100; x++) {  
    t = &array[x][0];  
    for (y = 0; y < 100; y++)  
        t[y] = func(x, y);  
}
```

This change saves the computation of `&array[x][0]` each time around the inner loop.

Strength Reduction

Strength reduction replaces an expensive operation such as multiplication with a less expensive operation such as addition. This replacement is performed when the loop index variable is found in a multiplicative expression. For example:

```
for (i = 0; i < 100; i++)  
    func(i * 5);
```

This could be replaced by:

```
t1 = 0;  
t2 = 5;  
for (i = 0; i < 100; i++) {  
    func(t1);  
    t1 += t2;  
}
```

The multiplication was replaced by a faster addition.

Loop Unrolling

Loop unrolling occurs when execution speed is of greater importance than code space. In this case, the compiler expands loops to repeated copies of the loop body. Although this adds to the code size, it eliminates the use of an index variable and the need to test its value repeatedly. For example:

```
for (i = 0; i < 5; i++)  
    func(i * 10);
```

This could be expanded to:

```
func(0);  
func(10);  
func(20);  
func(30);  
func(40);
```

Partial loop unrolling is also used. This reduces the number of loop iterations by generating more copies of the loop body. Partial loop unrolling saves increments and tests of the index variable.

For example:

```
for (i = 0; i < 100; i++)  
    func(i);
```

This could be replaced by the following:

```
for (i = 0; i < 100; i += 5) {  
    func(i);  
    func(i + 1);  
    func(i + 2);  
    func(i + 3);  
    func(i + 4);  
}
```

Constant Sharing

Ultra C examines each of the code area constants declared by a program and removes duplicates. This is most beneficial when an entire application is I-code linked so that the I-code optimizer sees all code area constants. For example:

`func1.c` contains:

```
char *func1()
{
    return find_verb("The last sandbag was placed on the levee at two o'clock.");
}
```

`func2.c` contains:

```
char *func2()
{
    return find_noun("The last sandbag was placed on the levee at two o'clock.");
}
```

Examining a program containing `func1.c` and `func2.c`, only one copy of the string used in both functions exists after I-code linking.

This optimization reduces the size of the program without affecting execution speed.

Function Inlining

When execution speed is of greater importance than code space, the compiler may move functions to the function from which they are called. Although this can add another copy of the function, it saves the overhead related to the function call and exit. If a function is only called once, inlining may reduce code size and improve execution time. The following is an example of function inlining:

Before inlining and optimizing:

```
main()
{
    int a, b, c, m;
    a = func(1);
    b = func(2);
    c = func(3);
    m = max(a, b, c);
    printf("max is %d\n",m);
}

int max(int x, int y,
        int z)
{
    if (x > y && x > z)
        return x;
    else if (y > z)
        return y;
    else
        return z;
}
```

After inlining and optimizing:

```
main()
{
    int a, b, c, m, t;
    a = func(1);
    b = func(2);
    c = func(3);
    if (a > b && a > c)
        t = a;
    else if (b > c)
        t = b;
    else
        t = c;
    m = t;
    printf("max is %d\n",m);
}
```

Function inlining is performed automatically on each file that the I-code optimizer optimizes. Only the safe (no code size increases) and trivial (a function so small that the calling overhead out-weighs the body) function inlines are performed without additional options from the command line.

If time/weight considerations are increased, more and larger functions are inlined regardless of significant increase in code size.

- The I-code optimizer `-m` option enables discarding of functions after inlining the function into every caller (assuming that reference to the function does not occur later in the compilation process).
- The I-code optimizer `-n` option disables inlining.

Span Dependent Optimizations

Span-dependent optimizations minimize the size of instructions containing instruction-relative displacements.



For More Information

Refer to the appropriate processor chapter in the *Ultra C/C++ Processor Guide* for additional span dependent optimization information.

Assembly Level Optimizations

The assembly level optimizations include:

- **Merging Common Tails**
- **Reducing Branch Complexity**
- **Location Tracking**
- **Offsetting Stack Changes**
- **Pipeline Scheduling**

Merging Common Tails

A **tail** is defined as a sequence of instructions ending in an unconditional branch instruction. If two tails contain the same instructions, they are redundant and one of them may be removed. The removed tail is replaced with a branch to the top of the other common tail. This optimization reduces code size and improves execution speed. For example, on the 68000, the following code:

```
move.l #-1,d0
unlk a5
movem.l d1-d7/a0-a4,(sp)+
rts
.
.
.
move.l #0,d0
unlk a5
movem.l d1-d7/a0-a4,(sp)+
rts
```

Could be changed to:

```
move.l #-1,d0
bra label1
.
.
```

```
.  
    move.l #0,d0  
label1  
    unlk a5  
    movem.l d1-d7/a0-a4,(sp)+  
    rts
```

Reducing Branch Complexity

The assembly optimizer simplifies branching constructs. Any unconditional branch that branches to another unconditional branch is changed to a branch to the final destination. A conditional branch that branches around an unconditional branch is changed to a simpler equivalent form. This optimization reduces code size and improves execution speed. For example:

```
    bcc label1  
    bra label2  
label1  
.  
.  
.
```

This could be changed to the following:

```
    bcs label2  
label1  
.  
.  
.
```

Location Tracking

The assembly code optimizer tracks the values placed in registers and memory and eliminates inefficient assembly language instructions. For example:

```
move.l a0,d0
move.l d0,a0
move.l d1,d6
move.l d1,d6
```

This may be changed to the following:

```
move.l a0,d0
move.l d1,d6
```

Offsetting Stack Changes

The assembly code optimizer eliminates offsetting changes to the stack pointer. This enables decreased code size and increased execution speed. For example, the following code sequence is eliminated:

```
add.l #8,sp
sub.l #8,sp
```

Pipeline Scheduling

The compiler rearranges instructions at the assembly language level so instructions that initialize a processor register are as far as possible from the use of the register. For example, on the 68040, a code sequence like:

```
move.l gptr(a6),a0
move.l (a0),d1
move.l #2,d0
bsr func
```

is changed to:

```
move.l gptra(a6),a0
move.l #2,d0
move.l (a0),d1
bsr func
```

In addition to these register use and define chains, floating-point instructions are mixed with integer instructions to more fully utilize instruction pipelines.

This optimization increases the speed of an application without affecting code size.

Chapter 12: Language Features



For More Information

Other implementation-defined areas are documented in the *Using Ultra C/C++ Processor Guide*, and the *Ultra C Library Reference* manual. Refer to the ANSI/ISO specification for more information.

In conformance with the ANSI/ISO specification, the implementation-defined areas of the compiler are listed in this chapter. Each item contains one implementation-defined issue. The number in parentheses included with each item indicates the location in the ANSI/ISO specification where more information may be found.

This chapter includes the following sections:

- **C Language Features**
- **C++ Language Features**



C Language Features

This section contains information on:

- **Translation**
- **Environment**
- **Identifiers**
- **Characters**
- **Integers**
- **Floating Point**
- **Structures, Unions, Enumerations, and Bit-Fields**
- **Qualifiers**
- **Declarators**
- **Statements**
- **Preprocessing Directives**

Translation

How a diagnostic is identified (5.1.1.3)

The following is the basic format of a diagnostic:

```
"<file>", line <n>: **** <diagnostic> ****  
<line>  
<circumflex>
```

<file> is the name of the file where the problem was encountered.

<n> is the line number on which the problem was encountered.

<diagnostic> is a message about the nature of the problem. If <diagnostic> starts with the text `warning -`, the problem is not fatal and compilation continues.

<line> is the ASCII text of the line that contains the problem.

<circumflex> is a circumflex (^) indicating the position in the line where the problem begins.

The following is an example of an error:

```
"test.c", line 3: **** undeclared identifier ****
printf("%d\n", var);
                ^
```

The following is an example of a warning:

```
"test.c", line 4: **** warning - 'return;' in
non-void function ****
return;
```

Environment

The semantics of the arguments to main (5.1.2.2.1).

The function definition for `main` may be written as:

```
int main(int argc, char *argv[], char *envp[])
```

`argc` contains the number of strings in the `argv` array. The strings pointed to by the elements of `argv` are the parameters specified by the process that forked the program.

`argv[1]` through `argv[argc - 1]` are generally the command line parameters from the shell command line.

`argv[0]` is, by convention, the name or pathlist of the file that contained the executing module. The `argv` array is terminated by a null pointer. That is, `argv[argc]` is `NULL`.

`envp` specifies the pointer to the environment.



Note

Passing the environment pointer as a third parameter is a common extension. However, it may not be portable.

What constitutes an interactive device (5.1.2.3).

Any device that uses one of the following file managers is considered interactive:

SCF	=	Sequential Character File Manager
Pipeman	=	Pipe File Manager
UCM	=	User Communication Manager
GFM	=	Graphics File Manager

Identifiers

The number of significant initial characters (beyond 31) in an identifier without external linkage (6.1.2).

The maximum number of characters in any identifier is 255. All characters are considered significant.

The number of significant initial characters (beyond 6) in an identifier with external linkage (6.1.2).

The maximum number of characters in any identifier is 255. All characters are considered significant.

Whether case distinctions are significant in an identifier with external linkage (6.1.2).

Case is significant in all identifiers.

Characters

The members of the source and execution character sets, except as explicitly specified in the standard (5.2.1).

The source character set is ASCII. However, SJIS characters are allowed within character constants, string literals, and comments. The execution character set is limited to ASCII characters.

The shift states used for the encoding of multibyte characters (5.2.1.2).

Shift states are not used in the encoding of multibyte characters.

The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.1.3.4).

Mapping is one to one.

The value of an integer character constant that contains more than one character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant (6.1.3.4).

Any value that fits in an integer is allowed. An error diagnostic is generated for out of range values.

The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character (6.1.3.4).

The value of an integer or multibyte character constant that contains more than one character is the value of the last character in the constant.

The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant (6.1.3.4).

The C locale with the addition of SJIS characters.

Integers

The results of bitwise operations on signed integers (6.3).

With the exception of right shift, the bitwise operators operate on signed values as if they were operating on unsigned values of the same width with the same bit pattern.

The result of a right shift of a negative-valued signed integral type (6.3.7).

A right shift (`>>`) of a signed integer causes the sign bit to propagate to the right. A right-shifted, negative-valued, signed integer remains negative-valued.

Floating Point

The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value (6.2.1.3).

The direction of truncation is to the nearest representable number with ties truncated to the lower number.

The direction of truncation or rounding when a floating-point number is converted to a narrower floating-point number (6.2.1.4).

The direction of truncation is to the nearest representable number with ties truncated to the lower number.

Structures, Unions, Enumerations, and Bit-Fields

A member of a union object is accessed using a member of a different type (6.3.2.3).

Because union objects overlap, accessing a union object of a different type causes the bit-pattern in the union to be interpreted as if it was of the different type.

Qualifiers

What constitutes an access to an object that has volatile-qualified type (6.5.5.3).

An access to a volatile-qualified object is when the object's value is required by the semantics of the code. Generally, an access (read or write) is generated for each use of the object.

Declarators

The maximum number of declarators that may modify an arithmetic, structure, or union type (6.5.4).

Thirteen declarations may modify an arithmetic, structure, or union type.

Statements

The maximum number of case values in a switch statement (6.6.4.2).

An infinite number of case values may be specified in a switch statement.

Preprocessing Directives

The behavior on each recognized `#pragma` directive (6.8.6).

The compiler has no recognized `#pragma` directives. Each use of the `#pragma` directive results in a warning.

The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (6.8.8).

If the date and time are not available at translation, `__DATE__` and `__TIME__` are replaced by `Jan 01 1970` and `00:00:00`, respectively.

C++ Language Features

This section contains information on:

- **Dialect Accepted**
- **Anachronisms Accepted**
- **Extensions Accepted**
- **Template Instantiation**
- **Instantiation Modes**
- **Instantiation #pragma Directives**
- **Implicit Inclusion**
- **Compiling with Exceptions Disabled**
- **Wide Character Strings**

Dialect Accepted

The front end accepts the C++ language as defined by *The Annotated C++ Reference Manual, (ARM)*, by Ellis and Stroustrup, Addison-Wesley, 1990, including templates, exceptions, and the anachronisms in this chapter. Some features have been updated to match the specification in the ANSI/ISO C++ Standards Committee **X3J16/WG21 Working Paper**.

Command-line options are also available to enable and disable anachronisms and strict standard-conformance checking.

The following features not in the ARM but in the X3J16/WG21 Working Paper are accepted:

- The dependent statement of an `if`, `while`, `do-while`, or `for` is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.

- The expression tested in an `if`, `while`, `do-while`, or `for`, as the first operand of a `?` operator, or as an operand of the `&&`, `||`, or `!` operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.
- A global-scope qualifier is allowed in member references of the form `x.::A::B` and `p->::A::B`.
- The precedence of the third operand of the `?` operator is changed.
- If control reaches the end of the `main()` routine, and `main()` has an integral return type, it is treated as if a `return 0;` statement were executed.
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form `A()` can be used even if `A` is a class without a (nontrivial) constructor. The temporary created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.
- A reference to `const volatile` cannot be bound to an rvalue.
- Qualification conversions such as conversion from `T**` to `T const * const *` are allowed.
- Digraphs are recognized.
- Operator keywords (such as `and`, `bitand`, etc.) are recognized.
- Static data member declarations can be used to declare member constants.

- `wchar_t` is recognized as a keyword and a distinct type.
- `bool` is recognized.
- RTTI (runtime type identification), including `dynamic_cast` and the `typeid` operator, is implemented.
- Declarations in tested conditions (in `if`, `switch`, `for`, and `while` statements) are supported.
- Array `new` and `delete` are implemented.
- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- `mutable` is accepted on nonstatic data member declarations.
- Namespaces are implemented, including `using` declarations and directives. Access declarations are broadened to match the corresponding `using` declarations.
- Explicit instantiation of templates is implemented.
- The `typename` keyword is recognized.
- `explicit` is accepted to declare non-converting constructors.
- The scope of a variable declared in the `for-init-statement` of a `for` loop is the scope of the loop (not the surrounding scope).
- Member templates are implemented.
- The new specialization syntax (using `template <>`) is implemented.
- Cv-qualifiers are retained on `rvalues` (in particular, on function return values).
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between PODs and non-PODs with trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).
- `extern inline` functions are supported, and the default linkage for `inline` functions is external.

- A `typedef` name may be used in an explicit destructor call.

The following features not in the ARM but in the X3J16/WG21 Working Paper are not accepted:

- Virtual functions in derived classes may not return a type that is the derived-class version of the type returned by the overridden function in the base class.
- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are not implemented.
- `enum` types cannot contain values larger than can be contained in an `int`.
- `reinterpret_cast` does not allow casting a pointer to member of one class to a pointer to member of another class if the classes are unrelated.
- Explicit qualification of template functions is not implemented.
- Name binding in templates in the style of N0288/93-0081 is not implemented.
- In a reference of the form `f()->g()`, with `g` a static member function, `f()` is not evaluated. This is as required by the ARM. The WP, however, requires that `f()` be evaluated.
- Class name injection is not implemented.
- Overloading of function templates (partial specialization) is not implemented.
- Partial specialization of class templates is not implemented.
- Placement delete is not implemented.
- Putting a `try/catch` around the initializers and body of a constructor is not implemented.
- The notation `:: template` (and `->template`, etc.) is not implemented.
- Template parameters are not implemented.
- Certain restrictions are not yet enforced on the use of (pointer-to-)function types that involve exception-specifications.

Anachronisms Accepted

The following anachronisms are accepted when anachronisms are enabled:

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array may be specified in an array `delete` operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.
- Assignment to `this` in constructors and destructors is allowed. This is allowed only if anachronisms are enabled and the “assignment to `this`” configuration parameter is enabled.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a non-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-`const` type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-`const` class type may be initialized from an `rvalue` of the class type or a derived class thereof. No (additional) temporary is used.

- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following declares the overloading of two functions named `f`:

```
int f(int);
int f(x) char x; { return x; }
```



Note

In C, this code is legal but has a different meaning: a tentative declaration of `f` is followed by its definition.

- When `--nonconst_ref_anachronism` is enabled, a reference to a non-const class can be bound to a class rvalue of the same type or a derived type thereof.

```
struct A {
    A(int);
    A operator=(A&);
    A operator+(const A&);
};
main () {
    A b(1);
    b = A(1) + A(2);    // Allowed as anachronism
}
```

Extensions Accepted

The following extensions are accepted in all modes (except when strict ANSI violations are diagnosed as errors):

- A friend declaration for a class may omit the `class` keyword:

```
class B;
class A {
    friend B;    // Should be "friend class B"
```

```
};
```

- Constants of scalar type may be defined within classes (this is an old form; the modern form uses an initialized static data member):

```
class A {
    const int size = 10;
    int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A {
    int A::f(); // Should be int f();
};
```

- `operator()` functions may have default argument expressions. A warning is issued.
- The preprocessing symbol `cplusplus` is defined in addition to the standard `__cplusplus`.
- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a default assignment operator — that is, such a declaration blocks the implicit generation of a copy assignment operator. (This is `cfront` behavior that is known to be relied upon in at least one widely used library.) Here's an example:

```
struct A { };
struct B : public A {
    B& operator=(A&);
};
```

- By default, as well as in `cfront`-compatibility mode, there is no implicit declaration of `B::operator=(const B&)`, whereas in strict-ANSI mode `B::operator=(A&)` is not a copy assignment operator and `B::operator=(const B&)` is implicitly declared.
- Extensions can also be enabled in `cfront`-compatibility mode or with command-line option `--implicit_extern_c_type_conversion`. Extensions are disabled in strict-ANSI mode.

Template Instantiation

The C++ language includes the concept of templates. A template is a description of a class or function that is a model for a family of related classes or functions. Since templates are descriptions of entities (typically, classes) that accept parameters according to the types they operate upon, they are sometimes called parameterized types. For example, you can write a template for a `Stack` class, and then use a stack of integers, a stack of floats, and a stack of some user-defined type. In the source, these might be written `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create instantiations of the template for each of the types required.

The instantiation of a class template is always done as soon as it is needed in a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (referred to as template entities) are not necessarily done immediately, for several reasons:

- You want to end up with only one copy of each instantiated entity across all the object files that make up a program. (This of course applies to entities with external linkage.)
- The language allows you to write a specialization of a template entity, for example, a specific version to be used in place of a version generated from the template for a specific data type. (You could, for example, write a version of `Stack<int>`, or of just `Stack<int>::push`, that replaces the template-generated version; often, such a specialization provides a more efficient representation for a particular data type.) Since the compiler cannot determine when compiling a reference to a template entity, if a specialization for that entity is provided in another compilation, it cannot do the instantiation automatically in any source file that references it. (The modern C++ language requires that a specialization be declared in every compilation in which it is used, but for compatibility with existing code and older compilers the Ultra C++ front end does not require that in some modes. See the command-line option `--no_distinct_template_signatures`.)

- The language also dictates that template functions that are not referenced should not be compiled. In fact, such functions might contain semantic errors that would prevent them from being compiled. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.



Note

Certain template entities are always instantiated when used, such as, inline functions.

From these requirements, you can see that if the compiler is responsible for doing all the instantiations automatically, it can only do so on a program-wide basis. The compiler cannot make decisions about instantiation of template entities until it has seen all the source files that make up a complete program.



For More Information

Chapter 8 describes automatic instantiation.

The Ultra C++ front end provides an instantiation mechanism that does automatic instantiation at link time. For cases where you want more explicit control over instantiation, the front end also provides instantiation modes and instantiation pragmas, which provide fine-grained control over the instantiation process.

Instantiation Modes

Normally, when a file is compiled, no template entities are instantiated, except those assigned to the file by automatic instantiation. The overall instantiation mode can, however, be changed by a command line option as defined in [Table 12-1](#).

Table 12-1 Instantiation Mode Command Line Options

Command	Description
<code>-tnone</code>	Do not automatically create instantiations of any template entities. This is the default. It is also the usually appropriate mode when automatic instantiation is done.
<code>-tused</code>	Instantiate those template entities that were used in the compilation. This includes all static data members for which there are template definitions.
<code>-tall</code>	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members are instantiated whether or not they were used. Nonmember template functions are instantiated even if the only reference was a declaration.
<code>-tlocal</code>	Similar to <code>-tused</code> except that the functions are given internal linkage. This provides a very simple mechanism for getting started with templates. The compiler instantiates the functions used in each compilation unit as local functions, and the program links and runs correctly (barring problems due to multiple copies of local static variables.) However, you may end up with many copies of the instantiated functions, so this is not suitable for production use. <code>-tlocal</code> cannot be used in conjunction with automatic template instantiation. If automatic instantiation is enabled by default, it is disabled by the <code>-tlocal</code> option. If automatic instantiation is not enabled by default, use of <code>-tlocal</code> and <code>-T</code> is an error.

Instantiation #pragma Directives

Instantiation pragmas can control the instantiation of specific template entities or sets of template entities. There are three instantiation pragmas:

- The `instantiate` pragma causes a specified entity to be instantiated.
- The `do_not_instantiate` pragma suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition is supplied.
- The `can_instantiate` pragma indicates that a specified entity can be instantiated in the current compilation, but need not be. It is used in conjunction with automatic instantiation, to indicate potential sites for instantiation if the template entity turns out to be required.



Note

The `can_instantiate` pragma forces the instantiation of the template instance even if it is not referenced somewhere else in the program.

Arguments to the instantiation pragma are identified in [Table 12-2](#).

Table 12-2 Instantiation Pragma Arguments

Instantiation Pragma	Argument
Template class name	<code>A<int></code>
Template class declaration	<code>class A<int></code>
Member function name	<code>A<int>::f</code>
Static data member name	<code>A<int>::i</code>

Table 12-2 Instantiation Pragma Arguments (continued)

Instantiation Pragma	Argument
Static data declaration	<code>int A<int>::i</code>
Member function declaration	<code>void A<int>::f(int, char)</code>
Template function declaration	<code>char* f(int, float)</code>

A pragma in which the argument is a template class name (such as, `A<int>` or `class A<int>`) is equivalent to repeating the pragma for each member function and static data member declared in the class. When instantiating an entire class, a given member function or static data member may be excluded using the `do_not_instantiate` pragma. For example,

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by the `instantiate` pragma and no template definition is available or a specific definition is provided, an error is issued.

```
template <class T> void f1(T);      * No body provided
template <class T> void g1(T);    * No body provided
void f1(int) {}                  * Specific definition
void main()
{
    int    i;
    double d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}
#pragma instantiate void f1(int)  * error - specific definition
#pragma instantiate void g1(int) * error - no body provided
```

`f1(double)` and `g1(double)` is not instantiated because no bodies were supplied. No errors are produced during the compilation if no bodies are supplied at link time, a linker error is produced.

A member function name (such as, `A<int>::f`) can only be used as a pragma argument if it refers to a single user-defined member function (for example, not an overloaded function). Compiler-generated functions are not considered, so a name may refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as in

```
#pragma instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation pragma may not be a compiler-generated function, an inline function, or a pure virtual function.

Implicit Inclusion

With implicit inclusion, the front end is given permission to assume that if it requires a definition to instantiate a template entity declared in a `.h` file it can implicitly include the corresponding `.c`, `.cc`, or `.cpp` file to get the source code for the definition. For example, a template entity `ABC::f` is declared in file `xyz.h`. And an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation. The compiler determines if the file `xyz.c`, `xyz.cc`, or `xyz.cpp` exists. If so, the compiler processes the file as if it was included at the end of the main source file.

To find the template definition file for a given template entity the front end requires the full path name of the file in which the template was declared, and whether the file was included using the system include syntax (such as, `#include <file.h>`). This information is not available for preprocessed source containing `#line` directives. Consequently, the front end does not attempt implicit inclusion for source code containing `#line` directives.

Implicit inclusion works well with automatic instantiation, but the two are independent. They can be enabled or disabled independently, and implicit inclusion is still useful when automatic instantiation is not done.

Implicit inclusions are only performed during the normal compilation of a file, (for example, not when doing only preprocessing). A common means of investigating certain kinds of problems is to produce a

preprocessed source file that can be inspected. When using implicit inclusion it is useful for the preprocessed source file to include any implicitly included files. This may be done using the `--no_preproc_only` command line option. This causes the preprocessed output to be generated as part of a normal compilation. When implicit inclusion is being used, the implicitly included files appear as part of the preprocessed output in the precise location at which they were included in the compilation.

Compiling with Exceptions Disabled

The Standard C++ library is now shipped in two versions: one compiled with exceptions enabled and one with exceptions disabled. The version with exceptions disabled is supplied as an object-code library `cpplibnx.l` and an i-code library `cpplibnx.il`. These libraries are linked in instead of the default libraries, if the `-qnx` option is used with `xcc`. When `-qnx` is used, the macro `__NO_EXCEPTIONS` is defined by the compiler, which can then be used to compile code conditionally. Microware's C++ headers are written such that they can be compiled with exceptions enabled or disabled.

Code compiled with exceptions enabled can be linked with code compiled with exceptions disabled as long as the regular version of the Standard Library is used, that is, no `-qnx` option is present on the command line which builds the program module. **Caveat:** it is possible that a `.ii` (instantiation information) file created during a compilation with exceptions disabled is no longer usable for a subsequent compilation with exceptions enabled (or vice versa), the most common error being multiple definitions of template entities. In this case, simply remove the `.ii` files and start again.

Exceptions Thrown from the Standard C++ Library

The Standard C++ library is required to throw exceptions in certain cases. Disabling exceptions prevents this from happening. Since the primary purpose of throwing exceptions from the Standard library is **notification** of error conditions, the exact effect is achieved by calling a user-defined exception handler function and passing it the object that would originally have been thrown if exceptions were enabled. Toward this goal, when exceptions are disabled, the Standard header `<exception>` contains the following additional types and function signatures:

```
namespace __mw {
    struct xinfo {
        const int line;
        const char* file;
        const std::exception& excep;
        xinfo(const char*, int, const std::exception&);
    };
    typedef void (*xhandler)(const xinfo&);
    extern xhandler set_exception(xhandler);
    extern void call_xhandler(int, const char *, const
std::exception&);
}
```

To handle any exception conditions occurring in the Standard library (as distinguished from actual thrown exceptions), the user can set the exception handler which is then called and passed a `__mw::xinfo` object as an argument. The various fields of the argument provide information about the "throw point" and the exception object which would have been thrown had exceptions been enabled.

Example

The `std::string` class reports accesses beyond the end of a string by throwing the exception `std::out_of_range`. With exceptions disabled, we can still get notification of this error condition as follows:

```
// assumes compiler option -qnx
#include <string>
#include <exception>
#include <iostream>
```

```
using __mw::xinfo;
using __mw::set_exception;
using std::cerr;
using std::endl;

extern void range_error_handler(const xinfo&);

int main()
{
    set_exception(&range_error_handler);
    string s = "abc";
    s.at(10) = 'Z'; // will cause std::out_of_range exception
}

void range_error_handler(const xinfo& xi)
{
    cerr << "exception: " << xi.excep.what() <<
        "file: " << xi.file << "line: " << xinfo.li << endl;
    abort();
}
```

The `excep` field of the argument `xi` to the handler is a reference to the original exception object that was required to be thrown when the error condition occurred in the Standard library. Since all exceptions thrown from the Standard library are derived from the base class `std::exception`, the virtual member function `what()` or the `typeid` operator can be used to find the actual type of the exception.

While this scheme allows notification of error conditions in the Standard C++ library, it does not, in general, allow the flexibility that full exception handling would. Typically, the only action that the exception handler function takes is to notify the user and abort the program.



WARNING

The default exception handler function just aborts the program via the Standard C library call `abort()` with no notification of any kind.

Exceptions Thrown from C++ Operations

Certain C++ operations like `dynamic_cast`, `new`, and `typeid` are required to throw exceptions implicitly (without an explicit `throw-expression`) when error conditions occur. With exceptions disabled, these error conditions are reported in a manner similar to that presented above, the difference being that the file and line information for the "throw point" are unavailable.

Example

Consider failure of `dynamic_cast`:

```
#include <exception>
struct Base { virtual ~Base() { } };
struct Derived : public Base { };

static volatile bool dynamic_cast_failed = false;
using __mw::xinfo;
using __mw::set_exception;
void dcast_fail_handler(const xinfo&);

int main()
{
    Base bobj;
    Base& bref = bobj;
    Derived& dref =
        dynamic_cast<Derived&>(bref); // will fail
    if (dynamic_cast_failed) {
        //...
    }
}
void dcast_fail_handler(const xinfo&)
{
    dynamic_cast_failed = true;
}
```

When memory allocation failure occurs and no `new_handler` is defined, the language requires throwing of a `std::bad_alloc` exception. With exceptions disabled, this can no longer be done, so the current exception handler is called as outlined above. No null pointer is returned unless the `nothrow` version of operator `new` is used. However, when exceptions are disabled, it is better to define and set a `new_handler` function.

Changes to Rogue Wave Tools++ Libraries

The Rogue Wave Tools++ libraries are also provided in "nx" versions now. The Rogue Wave Tools++ code does **not** throw exceptions, even when these are enabled. The reason is non-conformance with the C++ Standard with regard to exception and header names. As Rogue Wave updates their code to conform to the Standard, exception-enabled versions of the Rogue Wave Tools++ libraries will become available. The only reason for providing the "nx" versions in the current release is for space/time considerations. Currently, the mechanism to handle RW Tools exceptions is to set an error handler (in a manner similar to that for `cplib`). See the Rogue Wave Tools++ documentation for details.

Writing "Exception-Independent" Code

Ultra C++ allows you to write code which can compile correctly irrespective of whether exceptions are enabled or disabled. The techniques shown below can be used to write such code.

When exceptions are disabled, all C++ constructs related to exceptions can no longer be used. This includes exception specifications too. For example,

```
void f() throw();
```

can no longer be compiled, since `f()` is declared to not throw any exceptions, and this implies run time checking to make sure that any action that `f()` takes does not actually result in an exception. Also, you may wish to find out the overhead of using C++ exceptions but avoid writing two versions of the same code. You can accomplish this by using the pre-defined macro `__NO_EXCEPTIONS` and defining a few of your own:

```
#include <exception>
#ifdef __NO_EXCEPTIONS
#   define _Try try
#   define _Catch_exception      catch (::std::exception& _E)
#   define _Catch_bad_alloc     catch (::std::bad_alloc& _E)
#   define _Catch_bad_cast      catch (::std::bad_cast& _E)
#   define _Catch_bad_exception catch (::std::bad_exception& _E)
#   define _Catch_all           catch (...)
#   define _Ename                _E.what()
#   define _Throw0                throw () // exception specification
#   define _Throw(e)              throw e // throw expression
#endif
```

```

#else /*__NO_EXCEPTIONS*/
#  define _Try                if (true)
#  define _Catch_exception    if (false)
#  define _Catch_bad_alloc    if (false)
#  define _Catch_bad_cast     if (false)
#  define _Catch_bad_exception if (false)
#  define _Catch_all          if (false)
#  define _Ename              " "
#  define _Throw0
#  define _Throw(e) \
        __mw::call_xhandler(__FILE__, __LINE__, e)
#endif /*__NO_EXCEPTIONS*/

```

Since the most common (and useful) exception specification is the empty exception specification, whenever that needs to be specified, use the macro `_Throw0`. For example:

```
void f() _Throw0; // f() does not throw
```

Also, since `std::exception` is the base class of all standard exception types, whenever an exception is needed to be thrown from user code, make sure that the exception type is derived from `std::exception` and the virtual function `what()` is overridden appropriately. This way, catching a `const std::exception&` allows catching of all exceptions whose types are derived from `std::exception`. We can then use the macro `_Catch_exception` to mean `catch (const std::exception&)` when exceptions are turned on and `if (false)` when exceptions are turned off. The macro `_Try` evaluates to the keyword `try` with exceptions turned on and to `if (true)` otherwise. The macro `_Throw(ex)` throws the exception object `ex` when exceptions are enabled and calls `__mw::call_xhandler()` otherwise. An example using the above macros is shown below:

```

#include <exception>
/* ... above macros here or in an included header file ... */
class Array_bound_error : public std::exception {
    /*...*/
    virtual const char *what() const {return "index out of bounds";}
}
template<class T, int N> class Bounded_array {
    /* ... */
    T& operator[] (int i)
    {
        if (i < 0 || i >= N) _Throw(Array_bound_error());
        else
            /* ... */
    }
}

```

```
    }  
};  
  
#include <iostream>  
  
#ifdef __NO_EXCEPTIONS  
void X_handler(const __mw::xinfo& e)  
{  
    std::cerr << "file " << e.file << " line " <<  
        e.line << " exception " << e.excep.what() << std::endl;  
}  
#endif  
  
int main()  
{  
#ifdef __NO_EXCEPTIONS  
    __mw::set_exception(&X_handler);  
#endif  
  
    _Try {  
        Bounded_array<int, 10> arr;  
        arr[11] = 0;    // cause exception  
    }  
    _Catch_exception {  
        std::cerr << "exception: " << _Ename << std::endl;  
    }  
    _Catch_all {  
        std::cerr << "unknown exception!" << std::endl;  
    }  
}
```

Wide Character Strings

Strings of wide types (for example `wstring`) are not available by default. This is to prevent unnecessary code bloat.

If you want to use the `wstring` class, you must define `wstring` as the `wchar_t` version of the `basic_string` template.

This can be done with the following typedef:

```
#include <string>
namespace std {
    typedef basic_string<wchar_t> wstring;
}
```

At which point the `wstring` class is available:

```
wstring ws = L"wider is better";
```

Appendix A: Messages

This appendix defines messages generated by the compiler. Messages are categorized by the compiler component generating the message:

- **Executive**
- **Front End**
- **I-Code Linker**
- **I-Code Optimizer**
- **Back End**
- **Assembly Optimizer**
- **Assembler**
- **Prelinker**
- **Object Code Linker**



Executive

',' expected for -W

The phase in which to pass additional arguments must be delimited from the additional arguments by a comma in `c89` option mode.

****** stack overflow ******

The executive has run out of stack space on the host system. Use the host environment method to increase the stack space for an executable to increase the stack limit for the executive.

****** can't install cs1 ******

The `cs1` trap handler on the host system (68K or OS-9) is either missing or not in memory. Refer to the installation documentation for information on using `cs1`.

****** cs1 traphandler mismatch ******

The `cs1` trap handler installed on the system is out-of-date for the executive. If a new copy of Ultra C/C++ was recently installed, ensure that the `cs1` in memory is the one shipped with the new copy of Ultra C/C++.

-k=...f on 68020 and above only

The floating-point coprocessor option can only be used if the target processor is 68020 or greater.

-z input incomplete

The input file for the `-z` option in `c89` option mode ended with an option that requires an argument, but did not contain the argument for the option.

<file>: no recognized suffix

A file was named on the command line that does not end in a valid extension. Valid extensions vary for the different option modes.

abort/error result in '<file>'

The phase that was generating the output file returned a non-successful status. The executive renames the output file to `<file>` so that make utilities are not misled by the existence of an incomplete file.

argument expected for <string>

In `c89` option mode, an option was used that requires an argument and one was not supplied. Refer to the help text for `c89` option mode for more information on command line options.

assembly and/or back end output files not allowed when using data area layout

This error message may occur when the data area layout option is used. The back end must be able to process all files to be compiled to generate the executable. If assembly or back end output files are on the command line in this mode, the back end does not process them.

bad -k option — <string>

The option `<string>` is not supported in the `-k` argument in compatibility option mode.

can't allocate memory

The executive has requested memory from the host operating system and it is unavailable.

can't find compiler library file '<name>'

The standard compiler `library <name>` was not found. The verbose and dry run options used together print the `#include` and library file search paths.

can't find library file '<file>'

`<file>` cannot be found in any of the directories searched for library files. The verbose and dry run options used together print the `#include` and library file search paths.

can't open '<file>' for -z input

The file specified to read for additional arguments cannot be opened. More information about the type of error that occurred is provided by the host environment.

can't produce ROF when compiling to assembly language

In compatibility option mode, the executive generates this error when the `-r` option is used with the `-a` option.

illegal -O parameter

The argument for the `-O` option in `c89` option mode either contains non-decimal digit characters or is out of the range zero to seven.

illegal memory size specification

The addition stack space value is syntactically incorrect.

illegal optimization level

The optimization level specified is out of the range zero to seven.

invalid -W phase specifier — <char>

`<char>` does not specify a valid phase. Refer to help text in `c89` mode for more information on phase abbreviations.

invalid 68K or 020 options

The target processor sub-options in the Microware K&R C compiler are syntactically incorrect.

invalid additional stack size for phase

The additional stack space for a phase contains non-decimal digit characters or is zero.

invalid endpoint specified

`o1` cannot be specified as an endpoint as it is the default endpoint.

invalid numeric value for -o

The optimization level specified contains non-decimal digits.

invalid option mode — "<string>"

`<string>` does not properly specify an option mode. Only `ucc`, `compat`, or `c89` are allowed.

invalid phase for -x

Only `il`, `io`, and `ao` phases may be disabled using the `-x` option.

invalid phase for stack space option

The phase specified to have an enlarged stack is invalid. This error may also occur when the `-mode` option is out-of-place. `-mode` must be the first argument, before any options or file names specified on the command line as it determines the format of the remaining arguments.

invalid processor sub-options — <string>

The target processor sub-options are invalid. Refer to the help text for target processor sub-options for more information.

invalid target OS — <string>

The target OS specified is not valid. See the target operating system help option output for information on valid operating systems.

invalid target processor argument — <string>

The target processor specified in `c89` or `ucc` option mode is incorrect. Refer to the help text for target processor options for more information.

only one C or I-code file is allowed when using data area layout and the I-code linker is disabled

This error message may occur when the data area layout option is used. The back end must be able to process all files that are to be compiled to generate the executable. The back end can only process a single I-code file at a time.

output file name not applicable

When compiling to an ROF in the Microware K&R C compiler, an object code linker output file name is not allowed.

read error, -z input

The executive has called the operating system to read from the file specified with the `-z` option and received an error.

syntax error in <time or space> option argument

The time and space options are checked to ensure that they contain only decimal digits and that they are in the range zero to ten.

unknown option '<char>'

`<char>` is not a valid option in the selected option mode. The help for the current option mode is printed with this message for reference.

unsupported target OS/processor combination

The specified target OS and processor combination is invalid.

warning — <time or space> option over maximum, set to <num>

This warning indicates that the weight placed either time or space was too large and has been changed by the executive to <num>. To override this behavior when a thorough understanding of the effects is achieved, pass higher time and/or space options to specific phases. Refer to [Chapter 5: Compiler Phase Options](#), for more information.

warning: can't rename '<old>' to '<new>'

The executive called the operating system to rename the file <old> to <new> and was unsuccessful. The host environment should provide more information about the exact cause of the error. See also "abort/error result in '<file>'."

Front End

Front end messages consist of diagnostic and general messages.

Diagnostic messages have an associated **severity**, as follows:

- Catastrophic errors indicate problems of such severity that the compilation cannot continue, for example: command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one cannot be compiled.
- Errors indicate violations of the syntax or semantic rules of the C or C++ language. Compilation continues, but object code is not generated.
- Warnings indicate something valid but questionable. Compilation continues and object code is generated (if no errors are detected).
- Remarks indicate something that is valid and probably intended, but which a careful programmer may want to check. These diagnostics are not issued by default. Compilation continues and object code is generated (if no errors are detected).

Diagnostics are written to `stderr` with a form like the following:

```
"test.c", line 5: a break statement may only be used  
within a loop or switch break;  
    ^
```

Note that the message identifies the file and line involved, and that the source line itself (with position indicated by the `^`) follows the message. If there are several diagnostics in one source line, each diagnostic has the form above, with the result that the text of the source line is displayed several times, with an appropriate position each time.

Long messages are wrapped to additional lines when necessary.

A command line option may be used to request a shorter form of the diagnostic output in which the original source line is not displayed and the error message text is not wrapped when too long to fit on a single line.

A command line option may be used to request that the error number be included in the diagnostic message. When displayed, the error number also indicates whether the error may have its severity overridden on the command line. If the severity may be overridden, the error number includes the suffix `-D` (for discretionary); otherwise no suffix is present.

```
"Test_name.c", line 7: error #64-D: declaration does
not declare anythingstruct {};
    ^
```

```
"Test_name.c", line 9: error #77: this declaration
has no storage class or type specifier xxxxx;
    ^
```

Because an error is determined to be discretionary based on the error severity associated with a specific context, a given error may be discretionary in some cases and not in others.

For some messages, a list of entities is useful; they are listed following the initial error message:

```
"test.c", line 4: error: more than one instance of
overloaded function "f"
  matches the argument list:
  function "f(int)"
  function "f(float)"
  f(1.5);
  ^
```

In some cases, some additional context information is provided; specifically, such context information is useful when the front end issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

```
"test.c", line 7: error: "A::A()" is inaccessible B x;
                                     ^
  detected during implicit generation of "B::B()" at line 7
```

Without the context information, it is very hard to figure out what the error refers to.

The following diagnostic messages are issued:

- 0001 Last line of file ends without a newline.
- 0002 Last line of file ends with a backslash.
- 0003 `#include` file "xxxx" includes itself.
- 0004 Out of memory.
- 0005 Could not open source file "xxxx".
- 0006 Comment unclosed at end of file.
- 0007 Unrecognized token.
- 0008 Missing closing quote.
- 0009 Nested comment is not allowed.
- 0010 "#" not expected here.
- 0011 Unrecognized preprocessing directive.
- 0012 Parsing restarts here after previous syntax error.
- 0013 Expected a file name.
- 0014 Extra text after expected end of preprocessing directive.
- 0015 "xxxx" is not a file containing source text.
- 0016 "xxxx" is not a valid source file name.
- 0017 Expected a "]"
- 0018 Expected a ")"
- 0019 Extra text after expected end of number.
- 0020 Identifier "xxxx" is undefined.
- 0021 Type qualifiers are meaningless in this declaration.
- 0022 Invalid hexadecimal number.
- 0023 Integer constant is too large.
- 0024 Invalid octal digit.
- 0025 Quoted string should contain at least one character.
- 0026 Too many characters in character constant.

0027	Character value is out of range
0028	Expression must have a constant value.
0029	Expected an expression.
0030	Floating constant is out of range.
0031	Expression must have integral type.
0032	Expression must have arithmetic type.
0033	Expected a line number.
0034	Invalid line number.
0035	<code>#error</code> directive: <code>xxxx</code> .
0036	The <code>#if</code> for this directive is missing.
0037	The <code>#endif</code> for this directive is missing.
0038	Directive is not allowed — an <code>#else</code> has already appeared.
0039	Division by zero.
0040	Expected an identifier.
0041	Expression must have arithmetic or pointer type.
0042	Operand types are incompatible (" <code>type</code> " and " <code>type</code> ").
0044	Expression must have pointer type.
0045	<code>#undef</code> may not be used on this predefined name.
0046	This predefined name may not be redefined.
0047	Macro redefined differently.
0049	Duplicate macro parameter name.
0050	" <code>##</code> " may not be first in a macro definition.
0051	" <code>##</code> " may not be last in a macro definition.
0052	Expected a macro parameter name.
0053	Expected a " <code>:</code> ".
0054	Too few arguments in macro invocation.
0055	Too many arguments in macro invocation.

- 0056 Operand of `sizeof` may not be a function.
- 0057 This operator is not allowed in a constant expression.
- 0058 This operator is not allowed in a preprocessing expression.
- 0059 Function call is not allowed in a constant expression.
- 0060 This operator is not allowed in an integral constant expression.
- 0061 Integer operation result is out of range.
- 0062 Shift count is negative.
- 0063 Shift count is too large.
- 0064 Declaration does not declare anything.
- 0065 Expected a ";".
- 0066 Enumeration value is out of "int" range.
- 0067 Expected a "}".
- 0068 Integer conversion resulted in a change of sign.
- 0069 Integer conversion resulted in truncation.
- 0070 Incomplete type is not allowed.
- 0071 Operand of `sizeof` may not be a bit field.
- 0075 Operand of "*" must be a pointer.
- 0076 Argument to macro is empty.
- 0077 This declaration has no storage class or type specifier.
- 0078 A parameter declaration may not have an initializer.
- 0079 Expected a type specifier.
- 0080 A storage class may not be specified here.
- 0081 More than one storage class may not be specified.
- 0082 Storage class is not first.
- 0083 Type qualifier specified more than once.
- 0084 Invalid combination of type specifiers.
- 0085 Invalid storage class for a parameter.

0086	Invalid storage class for a function.
0087	A type specifier may not be used here.
0088	Array of functions is not allowed.
0089	Array of void is not allowed.
0090	Function returning function is not allowed.
0091	Function returning array is not allowed.
0092	Identifier-list parameters may only be used in a function definition.
0093	Function type may not come from a typedef.
0094	The size of an array must be greater than zero.
0095	Array is too large.
0096	A translation unit must contain at least one declaration.
0097	A function may not return a value of this type.
0098	An array may not have elements of this type.
0099	A declaration here must declare a parameter.
0100	Duplicate parameter name
0101	"xxxx" has already been declared in the current scope.
0102	Forward declaration of <code>enum</code> type is nonstandard.
0103	Class is too large.
0104	<code>struct</code> or union is too large.
0105	Invalid size for bit field.
0106	Invalid type for a bit field.
0107	Zero-length bit field must be unnamed.
0108	Signed bit field of length 1
0109	Expression must have (pointer-to-) function type.
0110	Expected either a definition or a tag name.
0111	Statement is unreachable.
0112	Expected "while".

- 0113 This use of a default argument is nonstandard.
- 0114 Entity-kind "entity" was referenced but not defined.
- 0115 A continue statement may only be used within a loop.
- 0116 A break statement may only be used within a loop or switch.
- 0117 Non-void entity-kind "entity" (declared at line xxxx) should return a value.
- 0118 A void function may not return a value.
- 0119 Cast to type "type" is not allowed.
- 0120 Return value type does not match the function type.
- 0121 A case label may only be used within a switch.
- 0122 A default label may only be used within a switch.
- 0123 Case label value has already appeared in this switch.
- 0124 Default label has already appeared in this switch.
- 0125 Expected a "(".
- 0126 Expression must be an lvalue.
- 0127 Expected a statement.
- 0128 Loop is not reachable from preceding code.
- 0129 A block-scope function may only have extern storage class.
- 0130 Expected a "{".
- 0131 Expression must have pointer-to-class type.
- 0132 Expression must have pointer-to-struct-or-union type.
- 0133 Expected a member name.
- 0134 Expected a field name.
- 0135 Entity-kind "entity" has no member "xxxx".
- 0136 Entity-kind "entity" has no field "xxxx".
- 0137 Expression must be a modifiable lvalue.
- 0138 Taking the address of a register variable is not allowed.
- 0139 Taking the address of a bit field is not allowed.

- 0140 Too many arguments in function call.
- 0141 Unnamed prototyped parameters not allowed when body is present.
- 0142 Expression must have pointer-to-object type.
- 0143 Program too large or complicated to compile.
- 0144 A value of type "type" cannot be used to initialize an entity of type "type".
- 0145 Entity-kind "entity" may not be initialized.
- 0146 Too many initializer values.
- 0147 Declaration is incompatible with entity-kind "entity" (declared at line xxxx).
- 0148 Entity-kind "entity" has already been initialized.
- 0149 A global-scope declaration may not have this storage class.
- 0150 A type name may not be re-declared as a parameter.
- 0151 A typedef name may not be re-declared as a parameter.
- 0152 Conversion of nonzero integer to pointer.
- 0153 Expression must have class type.
- 0154 Expression must have struct or union type.
- 0155 Old-fashioned assignment operator.
- 0156 Old-fashioned initializer.
- 0157 Expression must be an integral constant expression.
- 0158 Expression must be an lvalue or a function designator.
- 0159 Declaration is incompatible with previous "entity" (declared at line xxxx).
- 0160 Name conflicts with previously used external name "xxxx".
- 0161 Unrecognized #pragma.
- 0163 Could not open temporary file "xxxx".
- 0164 Name of directory for temporary files is too long ("xxxx").
- 0165 Too few arguments in function call.

- 0166 Invalid floating constant
- 0167 Argument of type "type" is incompatible with parameter of type "type".
- 0168 A function type is not allowed here.
- 0169 Expected a declaration.
- 0170 Pointer points outside of underlying object.
- 0171 Invalid type conversion.
- 0172 External/internal linkage conflict with previous declaration.
- 0173 Floating-point value does not fit in required integral type.
- 0174 Expression has no effect.
- 0175 Subscript out of range.
- 0177 Entity-kind "entity" was declared but never referenced.
- 0178 "&" applied to an array has no effect.
- 0179 Right operand of "%" is zero.
- 0180 Argument is incompatible with formal parameter.
- 0181 Argument is incompatible with corresponding format string conversion.
- 0182 Could not open source file "xxxx" (no directories in search list).
- 0183 Type of cast must be integral.
- 0184 Type of cast must be arithmetic or pointer.
- 0185 Dynamic initialization in unreachable code.
- 0186 Pointless comparison of unsigned integer with zero.
- 0187 Possible use of "=" where "==" was intended.
- 0188 Enumerated type mixed with another type.
- 0189 Error while writing xxxx file.
- 0190 Invalid intermediate language file.
- 0191 Type qualifier is meaningless on cast type.

- 0192 Unrecognized character escape sequence.
- 0193 Zero used for undefined preprocessing identifier.
- 0194 Expected an `asm` string.
- 0195 An `asm` function must be prototyped.
- 0196 An `asm` function may not have an ellipsis.
- 0219 Error while deleting file "xxxx".
- 0220 Integral value does not fit in required floating-point type.
- 0221 Floating-point value does not fit in required floating-point type.
- 0222 Floating-point operation result is out of range.
- 0223 Function declared implicitly.
- 0224 The format string requires additional arguments.
- 0225 The format string ends before this argument.
- 0226 Invalid format string conversion.
- 0227 Macro recursion.
- 0228 Trailing comma is nonstandard.
- 0229 Bit field cannot contain all values of the enumerated type.
- 0230 Nonstandard type for a bit field.
- 0231 Declaration is not visible outside of function.
- 0232 Old-fashioned `typedef` of "void" ignored.
- 0233 Left operand is not a `struct` or union containing this field.
- 0234 Pointer does not point to `struct` or union containing this field.
- 0235 Variable "xxxx" was declared with a never-completed type.
- 0236 Controlling expression is constant.
- 0237 Selector expression is constant.
- 0238 Invalid specifier on a parameter.
- 0239 Invalid specifier outside a class declaration.

- 0240 Duplicate specifier in declaration.
- 0241 A union is not allowed to have a base class.
- 0242 Multiple access control specifiers are not allowed.
- 0243 Class or `struct` definition is missing.
- 0244 Qualified name is not a member of class "type" or its base classes.
- 0245 A nonstatic member reference must be relative to a specific object.
- 0246 A nonstatic data member may not be defined outside its class.
- 0247 Entity-kind "entity" has already been defined.
- 0248 Pointer to reference is not allowed.
- 0249 Reference to reference is not allowed.
- 0250 Reference to void is not allowed.
- 0251 Array of reference is not allowed.
- 0252 Reference entity-kind "entity" requires an initializer.
- 0253 Expected a ", ".
- 0254 Type name is not allowed.
- 0255 Type definition is not allowed.
- 0256 Invalid re-declaration of type name "entity" (declared at line `xxxx`).
- 0257 Const entity-kind "entity" requires an initializer.
- 0258 "this" may only be used inside a nonstatic member function.
- 0259 Constant value is not known.
- 0260 Explicit type is missing ("int" assumed).
- 0261 Access control not specified ("`xxxx`" by default).
- 0262 Not a class or `struct` name.
- 0263 Duplicate base class name.
- 0264 Invalid base class.

- 0265 Entity-kind "entity" is inaccessible.
- 0266 "entity" is ambiguous.
- 0267 Old-style parameter list (anachronism).
- 0268 Declaration may not appear after executable statement in block.
- 0269 Implicit conversion to inaccessible base class "type" is not allowed.
- 0274 Improperly terminated macro invocation.
- 0276 Name followed by "::" must be a class or namespace name.
- 0277 Invalid friend declaration.
- 0278 A constructor or destructor may not return a value.
- 0279 Invalid destructor declaration.
- 0280 Invalid declaration of a member with the same name as its class.
- 0281 Global-scope qualifier (leading "::") is not allowed.
- 0282 The global scope has no "xxxx".
- 0283 Qualified name is not allowed.
- 0284 NULL reference is not allowed.
- 0285 Initialization with "{...}" is not allowed for object of type "type".
- 0286 Base class "type" is ambiguous.
- 0287 Derived class "type" contains more than one instance of class "type".
- 0288 Cannot convert pointer to base class "type" to pointer to derived class "type" — base class is virtual.
- 0289 No instance of constructor "entity" matches the argument list.
- 0290 Copy constructor for class "type" is ambiguous.
- 0291 No default constructor exists for class "type".

- 0292 "xxxx" is not a nonstatic data member or base class of class "type".
- 0293 Indirect nonvirtual base class is not allowed.
- 0294 Invalid union member — class "type" has a disallowed member function.
- 0295 Cannot overload functions — parameter types are too similar.
- 0296 Invalid use of non-lvalue array.
- 0297 Expected an operator.
- 0298 Inherited member is not allowed.
- 0299 Cannot determine which instance of entity-kind "entity" is intended.
- 0300 A pointer to a bound function may only be used to call the function.
- 0301 Typedef name has already been declared (with same type).
- 0302 Entity-kind "entity" has already been defined.
- 0304 No instance of entity-kind "entity" matches the argument list.
- 0305 Type definition is not allowed in function return type declaration.
- 0306 Default argument not at end of parameter list.
- 0307 Redefinition of default argument.
- 0308 More than one instance of entity-kind "entity" matches the argument list.
- 0309 More than one instance of constructor "entity" matches the argument list.
- 0310 Default argument of type "type" is incompatible with parameter of type "type".
- 0311 Cannot overload functions distinguished by return type alone.
- 0312 No suitable user-defined conversion from "type" to "type" exists.
- 0313 Type qualifier is not allowed on this function.

- 0314 Only nonstatic member functions may be virtual.
- 0315 The object has type qualifiers that are not compatible with the member function.
- 0316 Program too large to compile (too many virtual functions).
- 0317 Type differs from base class virtual function by return type alone.
- 0318 Override of virtual entity-kind "entity" is ambiguous.
- 0319 Pure specifier ("= 0") allowed only on virtual functions.
- 0320 Badly-formed pure specifier (only "= 0" is allowed).
- 0321 Data member initializer is not allowed.
- 0322 Object of abstract class type is not allowed.
- 0323 Function returning abstract class is not allowed.
- 0324 Duplicate friend declaration.
- 0325 Inline specifier allowed on function declarations only.
- 0326 "inline" is not allowed.
- 0327 Invalid storage class for an inline function.
- 0328 Invalid storage class for a class member.
- 0329 Local class member entity-kind "entity" requires a definition.
- 0330 Entity-kind "entity" is inaccessible.
- 0332 Class "type" has no copy constructor to copy a `const` object.
- 0333 Defining an implicitly declared member function is not allowed.
- 0334 Class "type" has no suitable copy constructor.
- 0335 Linkage specification is not allowed.
- 0336 Unknown external linkage specification.
- 0337 Linkage specification is incompatible with previous "entity" (declared at line `xxxx`).

- 0338 More than one instance of entity-kind "entity" has "c" linkage.
- 0339 Class "type" has more than one default constructor.
- 0340 Value copied to temporary, reference to temporary used.
- 0341 "operatorxxxx" must be a member function.
- 0342 Operator may not be a static member function.
- 0343 No arguments allowed on user-defined conversion.
- 0344 Too many arguments for operator function.
- 0345 Too few arguments for operator function.
- 0346 Nonmember operator requires an argument with class type.
- 0347 Default argument is not allowed.
- 0348 More than one user-defined conversion from "type" to "type" applies.
- 0349 No operator "xxxx" matches these operands.
- 0350 More than one operator "xxxx" matches these operands.
- 0351 First parameter of allocation function must be of type "size_t".
- 0352 Allocation function requires "void *" return type.
- 0353 Deallocation function requires "void" return type.
- 0354 First parameter of deallocation function must be of type "void *".
- 0355 Second parameter of deallocation function must be of type "size_t".
- 0356 Type must be an object type.
- 0357 Base class "type" has already been initialized.
- 0358 Base class name required — "type" assumed (anachronism).
- 0359 Entity-kind "entity" has already been initialized.
- 0360 Name of member or base class is missing.

- 0361 Assignment to "this" (anachronism).
- 0362 "overload" keyword used (anachronism).
- 0363 Invalid anonymous union — nonpublic member is not allowed.
- 0364 Invalid anonymous union — member function is not allowed.
- 0365 Anonymous union at global or namespace scope must be declared static.
- 0366 Entity-kind "entity" provides no initializer for.
- 0367 Implicitly generated constructor for class "type" cannot initialize.
- 0368 Entity-kind "entity" defines no constructor to initialize the following.
- 0369 Entity-kind "entity" has an uninitialized `const` or reference member.
- 0370 Entity-kind "entity" has an uninitialized `const` field.
- 0371 Class "type" has no assignment operator to copy a `const` object.
- 0372 Class "type" has no suitable assignment operator.
- 0373 Ambiguous assignment operator for class "type".
- 0375 Declaration requires a `typedef` name.
- 0377 "virtual" is not allowed.
- 0378 "static" is not allowed.
- 0379 Cast of bound function to normal function pointer (anachronism).
- 0380 Expression must have pointer-to-member type.
- 0381 Extra ";" ignored.
- 0382 Nonstandard member constant declaration.
- 0384 No instance of overloaded "entity" matches the argument list.
- 0385 Operator `delete()` may not be overloaded.

- 0386 No instance of entity-kind "entity" matches the required type.
- 0387 Delete array size expression used (anachronism).
- 0388 "type" is an invalid return type for "entity".
- 0389 A cast to an abstract class is not allowed.
- 0390 Function "main" may not be called or have its address taken.
- 0391 A new-initializer may not be specified for an array.
- 0392 Member function "entity" may not be redeclared outside its class.
- 0393 Pointer to incomplete class type is not allowed.
- 0394 Reference to local variable of enclosing function is not allowed.
- 0395 Single-argument function used for postfix "xxxx" (anachronism).
- 0397 Implicitly generated assignment operator cannot copy.
- 0398 Cast to array type is nonstandard (treated as cast to "type").
- 0399 Entity-kind "entity" has an operator newxxxx() but no operator deletexxxx().
- 0400 Entity-kind "entity" has an operator deletexxxx() but no operator newxxxx().
- 0401 Destructor for base class "type" is not virtual.
- 0402 Entity-kind "entity" has no accessible constructors.
- 0403 Entity-kind "entity" has already been declared.
- 0404 Function "main" may not be declared inline.
- 0405 Member function with the same name as its class must be a constructor.
- 0406 Using nested entity-kind "entity" (anachronism).
- 0407 A destructor may not have parameters.
- 0408 Copy constructor for class "type" may not have a parameter of type "type".

- 0409 Function return type is incomplete.
- 0410 Protected entity-kind "entity" is not accessible through a "type" pointer or object.
- 0411 A parameter is not allowed.
- 0412 An "asm" declaration is not allowed here.
- 0413 No suitable conversion function from "type" to "type" exists.
- 0414 Delete of pointer to incomplete class.
- 0415 No suitable constructor exists to convert from "type" to "type".
- 0416 More than one constructor applies to convert from "type" to "type".
- 0417 More than one conversion function from "type" to "type" applies.
- 0418 More than one conversion function from "type" to a built-in type applies.
- 0419 ec_const_member [const %n]
- 0420 ec_reference_member [reference %n]
- 0421 ec_ambiguous_function_add_on [%np]
- 0422 ec_builtin_operator_add_on [built-in operator %sq]
- 0423 ec_ambiguous_by_inheritance_add_on [%no (ambiguous by inheritance)]
- 0424 A constructor or destructor may not have its address taken.
- 0425 Dollar sign ("\$") used in identifier.
- 0426 Temporary used for initial value of reference to non-const (anachronism).
- 0427 Qualified name is not allowed in member declaration.
- 0428 Enumerated type mixed with another type (anachronism).
- 0429 The size of an array in "new" must be non-negative.
- 0430 Returning reference to local temporary.

- 0431 `Const` qualifier dropped in initializing reference to `non-const`.
- 0432 `"enum"` declaration is not allowed.
- 0433 Qualifiers dropped in binding reference of type `"type"` to initializer of type `"type"`.
- 0434 A reference of type `"type"` (not `const`-qualified) cannot be initialized with a value of type `"type"`.
- 0435 A pointer to function may not be deleted.
- 0436 Conversion function must be a nonstatic member function.
- 0437 Template declaration is not allowed here.
- 0438 Expected a `"<"`.
- 0439 Expected a `">"`.
- 0440 Template parameter declaration is missing.
- 0441 Argument list for `entity-kind "entity"` is missing.
- 0442 Too few arguments for `entity-kind "entity"`.
- 0443 Too many arguments for `entity-kind "entity"`.
- 0444 Template parameter for a function template must be a type.
- 0445 Entity-kind `"entity"` is not used in declaring the parameter types of `entity-kind "entity"`.
- 0446 Two nested types have the same name: `"entity"` and `"entity"` (declared at line `xxxx`). (`cfront` compatibility)
- 0447 Global `"entity"` was declared after nested `"entity"` (declared at line `xxxx`) (`cfront` compatibility).
- 0449 More than one instance of `entity-kind "entity"` matches the required type.
- 0450 The type `"long long"` is nonstandard.
- 0451 Omission of `"xxxx"` is nonstandard.
- 0452 Return type may not be specified on a conversion function.
- 0453 Template detected during heading.
- 0454 Instantiation of `%nf %p`.

- 0455 Implicit generation of `%nf %p`.
- 0456 Excessive recursion at instantiation of entity-kind "entity".
- 0457 "xxxx" is not a function or static data member.
- 0458 Argument of type "type" is incompatible with template parameter of type "type".
- 0459 Initialization requiring a temporary or conversion is not allowed.
- 0460 Declaration of "xxxx" hides function parameter.
- 0461 Initial value of reference to non-const must be an lvalue.
- 0462 Implicit definition of `%nf %p`.
- 0463 "template" is not allowed.
- 0464 "type" is not a class template.
- 0466 "main" is not a valid name for a function template.
- 0467 Invalid reference to entity-kind "entity" (union/nonunion mismatch).
- 0468 A template argument may not reference a local type.
- 0469 Tag kind of `xxxx` is incompatible with declaration of entity-kind "entity" (declared at line `xxxx`).
- 0470 The global scope has no tag named "xxxx".
- 0471 Entity-kind "entity" has no tag member named "xxxx".
- 0472 Member function `typedef` (allowed for `cfront` compatibility).
- 0473 Entity-kind "entity" may be used only in pointer-to-member declaration.
- 0475 A template argument may not reference a non-external entity.
- 0476 Name followed by "::~" must be a class name or a type name.
- 0477 Destructor name does not match name of class "type".
- 0478 Type used as destructor name does not match type "type".

- 0479 Entity-kind "entity" redeclared "inline" after being called.
- 0481 Invalid storage class for a template declaration.
- 0482 Entity-kind "entity" is an inaccessible type (allowed for cfront compatibility).
- 0483 A return type is not allowed.
- 0484 Invalid explicit instantiation declaration.
- 0485 Entity-kind "entity" is not an entity that can be instantiated.
- 0486 Compiler generated entity-kind "entity" cannot be instantiated.
- 0487 Inline entity-kind "entity" cannot be instantiated.
- 0488 Pure virtual entity-kind "entity" cannot be instantiated.
- 0489 Entity-kind "entity" cannot be instantiated -- no template definition was supplied.
- 0490 Entity-kind "entity" cannot be instantiated -- it has been explicitly specialized.
- 0491 Class "type" has no constructor.
- 0492 Entity-kind "entity" must be used in a parameter without a default value in entity-kind "entity".
- 0493 No instance of entity-kind "entity" matches the specified type.
- 0494 Declaring a void parameter list with a typedef is nonstandard.
- 0495 Global entity-kind "entity" used instead of entity-kind "entity" (cfront compatibility).
- 0496 Template parameter "xxxx" may not be re-declared in this scope.
- 0497 Declaration of "xxxx" hides template parameter.
- 0498 Template argument list must match the parameter list.

- 0499 Conversion function to convert from "type" to "type" is not allowed.
- 0500 Extra argument of postfix "operatorxxxx" must be of type "int".
- 0501 An operator name must be declared as a function.
- 0502 Operator name is not allowed.
- 0503 Entity-kind "entity" cannot be specialized in the current scope.
- 0504 Nonstandard form for taking the address of a member function.
- 0505 Too few template parameters — does not match previous declaration.
- 0506 Too many template parameters — does not match previous declaration.
- 0507 Function template for operator `delete()` is not allowed.
- 0508 Class template and template parameter may not have the same name.
- 0509 "entity" cannot be used to designate constructor for entity-kind "entity".
- 0510 A template argument may not reference an unnamed type.
- 0511 Enumerated type is not allowed.
- 0512 Type qualifier on a reference type is not allowed.
- 0513 A value of type "type" cannot be assigned to an entity of type "type".
- 0514 Pointless comparison of unsigned integer with a negative constant.
- 0515 Cannot convert to incomplete class "type".
- 0516 `Const` object requires an initializer.
- 0517 Object has an uninitialized `const` or reference member.
- 0518 Nonstandard preprocessing directive.

- 0519 Entity-kind "entity" may not have a template argument list.
- 0520 Initialization with "{...}" expected for aggregate object.
- 0521 Pointer-to-member selection class types are incompatible ("type" and "type").
- 0522 Pointless friend declaration.
- 0523 "." used in place of "::" to form a qualified name (cfront anachronism).
- 0524 Non-const function called for const object (cfront anachronism).
- 0525 A dependent statement may not be a declaration.
- 0526 A parameter may not have void type.
- 0527 Instantiation of %na %p.
- 0528 Processing of template argument list for %na %p.
- 0529 This operator is not allowed in a template argument expression.
- 0530 Try block requires at least one handler.
- 0531 Handler requires an exception declaration.
- 0532 Handler is masked by default handler.
- 0533 Handler is potentially masked by previous handler for type "type".
- 0534 Use of a local type to specify an exception.
- 0535 Redundant type in exception specification.
- 0536 Exception specification is incompatible with that of previous entity-kind "entity" (declared at line xxxxx).
- 0537 Previously specified: no exceptions will be thrown.
- 0538 Previously omitted: %t.
- 0539 Previously specified but omitted here: %t.
- 0540 Support for exception handling is disabled.

- 0541 Omission of exception specification is incompatible with previous entity-kind "entity" (declared at line `xxxx`).
- 0542 Could not create instantiation information file "`xxxxx`".
- 0543 Non-arithmetic operation not allowed in nontype template argument.
- 0544 Use of a local type to declare a nonlocal variable.
- 0545 Use of a local type to declare a function.
- 0546 Transfer of control bypasses initialization.
- 0547 Declare position: `%nd`.
- 0548 Transfer of control into an exception handler.
- 0549 Entity-kind "entity" is used before its value is set.
- 0550 Entity-kind "entity" was set but never used.
- 0551 Entity-kind "entity" cannot be defined in the current scope.
- 0552 Exception specification is not allowed.
- 0553 External/internal linkage conflict for entity-kind "entity" (declared at line `xxxx`).
- 0554 Entity-kind "entity" is not called for implicit or explicit conversions.
- 0555 Tag kind of `xxxx` is incompatible with template parameter of type "type".
- 0556 Function template for operator `new(size_t)` is not allowed.
- 0558 Pointer to member of type "type" is not allowed.
- 0559 Ellipsis is not allowed in operator function parameter list.
- 0560 "entity" is reserved for future use as a keyword.
- 0561 Invalid macro definition.
- 0562 Invalid macro undefinition.
- 0563 Invalid preprocessor output file.
- 0564 Cannot open preprocessor output file.

- 0565 `IL` file name must be specified if input is.
- 0566 Invalid `IL` output file.
- 0567 Cannot open `IL` output file.
- 0568 Invalid `C` output file.
- 0569 Cannot open `C` output file.
- 0570 Error in debug option argument.
- 0571 Invalid option.
- 0572 Back end requires name of `IL` file.
- 0573 Could not open `IL` file.
- 0574 Invalid number.
- 0575 Incorrect host CPU id.
- 0576 Invalid instantiation mode.
- 0578 Invalid error limit.
- 0579 Invalid raw-listing output file.
- 0580 Cannot open raw-listing output file.
- 0581 Invalid cross-reference output file.
- 0582 Cannot open cross-reference output file.
- 0583 Invalid cross-reference output file.
- 0584 Cannot open cross-reference output file.
- 0585 Invalid error output file.
- 0586 Cannot open error output file.
- 0587 Virtual function tables can only be suppressed when compiling C++.
- 0588 Anachronism option can be used only when compiling C++.
- 0589 Instantiation mode option can be used only when compiling C++.
- 0590 Automatic instantiation mode can be used only when compiling C++.

- 0591 Implicit template inclusion mode can be used only when compiling C++.
- 0592 Exception handling option can be used only when compiling C++.
- 0593 Strict ANSI mode is incompatible with K&R mode.
- 0594 Strict ANSI mode is incompatible with cfront mode.
- 0595 Missing source file name.
- 0596 Output files may not be specified when compiling several input files.
- 0597 Too many arguments on command line.
- 0598 An output file was specified, but none is needed.
- 0599 IL display requires name of IL file.
- 0600 A template parameter may not have void type.
- 0601 Excessive recursive instantiation of %n due to instantiate-all mode.
- 0602 Strict ANSI mode is incompatible with allowing anachronisms.
- 0603 A throw expression may not have void type.
- 0604 Local instantiation mode is incompatible with automatic instantiation.
- 0605 Parameter of abstract class type %t is not allowed.
- 0606 Array of abstract class %t is not allowed.
- 0607 Floating-point template parameter is nonstandard.
- 0608 This pragma must immediately precede a declaration.
- 0609 This pragma must immediately precede a statement.
- 0610 This pragma must immediately precede a declaration or statement.
- 0611 This kind of pragma may not be used here.
- 0612 %nf1 does not match %no2; virtual function override intended.

- 0613 Overloaded virtual function %n01 is only partially overridden in %n2.
- 0614 Specific definition of inline template function must precede its first use.
- 0615 Invalid error tag.
- 0616 Invalid error number.
- 0617 Parameter type involves pointer to array of unknown bound.
- 0618 Parameter type involves reference to array of unknown bound.
- 0619 Pointer-to-member-function cast to pointer to function.
- 0620 struct or union declares no named members.
- 0621 Nonstandard unnamed field.
- 0622 Nonstandard unnamed member.
- 0624 Invalid precompiled header output file.
- 0625 Cannot open precompiled header output file.
- 0626 %sq is not a type name.
- 0627 Cannot open precompiled header input file.
- 0628 Precompiled header file %sq is either invalid or not generated by this version of the Compiler.
- 0629 Precompiled header file %sq was not generated in this directory.
- 0630 Header files used to generate precompiled header file %sq have changed.
- 0631 The command line options do not match those used when precompiled header file %sq was created.
- 0632 The initial sequence of preprocessing directives is not compatible with those of precompiled header file %sq.
- 0633 Unable to obtain mapped memory.
- 0634 %s: using precompiled header file %s.
- 0635 %s: creating precompiled header file %s.

- 0636 Memory usage conflict with precompiled header file `%sq`.
- 0637 Invalid PCH memory size.
- 0638 PCH options must appear first in the command line.
- 0639 Insufficient memory for PCH memory allocation.
- 0640 Precompiled header files may not be used when compiling several input files.
- 0641 Insufficient preallocated memory for generation of precompiled header file (`%s` bytes required).
- 0642 Very large entity in program prevents generation of precompiled header file.
- 0643 `%sq` is not a valid directory.
- 0644 Cannot build temporary file name.
- 0645 `restrict` is not allowed.
- 0646 A pointer or reference to function type may not be qualified by `restrict`.
- 0647 `%sq` is an invalid `__declspec` attribute.
- 0648 A calling convention modifier may not be specified here.
- 0649 Conflicting calling convention modifiers.
- 0650 Strict ANSI mode is incompatible with Microsoft mode.
- 0651 `cfront` mode is incompatible with Microsoft mode.
- 0652 Calling convention specified here is ignored.
- 0653 A calling convention may not be followed by a nested declarator.
- 0654 Calling convention is ignored for this type.
- 0656 Declaration modifiers are incompatible with previous declaration.
- 0657 The modifier `%sq` is not allowed on this declaration.
- 0658 Transfer of control into a try block.
- 0659 Inline specification is incompatible with previous `%nod`.

- 0660 Closing brace of template definition not found.
- 0661 `wchar_t` keyword option can be used only when compiling C++.
- 0662 Invalid packing alignment value.
- 0663 Expected an integer constant.
- 0664 Call of pure virtual function.
- 0665 Invalid source file identifier string.
- 0666 A class template cannot be defined in a friend declaration.
- 0667 `asm` is not allowed.
- 0668 `asm` must be used with a function definition.
- 0669 `asm` function is nonstandard.
- 0670 Ellipsis with no explicit parameters is nonstandard.
- 0671 “&...” is nonstandard.
- 0672 Invalid use of “&...”.
- 0674 Temporary used for initial value of reference to const volatile (anachronism).
- 0675 A reference of type `%t1` cannot be initialized with a value of type `%t2`.
- 0676 Initial value of reference to `const volatile` must be an lvalue.
- 0677 SVR4 C compatibility option can be used only when compiling ANSI C.
- 0678 Using out-of-scope declaration of `%nd`.
- 0679 Strict ANSI mode is incompatible with SVR4 C mode.
- 0680 Call of `%nd` cannot be inlined.
- 0681 `%n` cannot be inlined.
- 0682 Invalid PCH directory.
- 0683 Expected `__except` or `__finally`.
- 0684 A `__leave` statement may only be used within a `__try`.

- 0685 Detected during instantiation of `%nf %p`.
- 0686 Detected during implicit generation of `%nf %p`.
- 0687 Detected during instantiation of `%na %p`.
- 0688 Detected during processing of template argument list for `%na %p`.
- 0689 Detected during implicit definition of `%nf %p`.
- 0690 `%sq` not found on pack alignment stack.
- 0691 Empty pack alignment stack.
- 0692 RTTI option can be used only when compiling C++.
- 0693 `%nf`, required for copy that was eliminated, is inaccessible.
- 0694 `%nf`, required for copy that was eliminated, is not callable because reference parameter cannot be bound to `rvalue`.
- 0695 `<typeinfo>` must be included before `typeid` is used.
- 0696 `%s` cannot cast away `const` or other type qualifiers.
- 0697 The type in a `dynamic_cast` must be a pointer or reference to a complete class type, or `void *`.
- 0698 The operand of a pointer `dynamic_cast` must be a pointer to a complete class type.
- 0699 The operand of a reference `dynamic_cast` must be an `lvalue` of a complete class type.
- 0700 The operand of a runtime `dynamic_cast` must have a polymorphic class type.
- 0701 `Bool` option can be used only when compiling C++.
- 0702 Invalid storage class for condition declaration.
- 0703 An array type is not allowed here.
- 0704 Expected an `"="`.
- 0705 Expected a declarator in condition declaration.
- 0706 `%sq`, declared in condition, may not be redeclared in this scope.

- 0707 Default template arguments are not allowed for function templates.
- 0708 Expected a “;” or “>”.
- 0709 Expected a template parameter list.
- 0710 Incrementing a bool value is deprecated.
- 0711 Bool type is not allowed.
- 0712 Offset of base class %no1 within class %no2 is too large.
- 0713 Expression must have bool type (or be convertible to bool).
- 0714 Array new and delete option can be used only when compiling C++.
- 0715 %n is not a variable name.
- 0716 __based modifier is not allowed here.
- 0717 __based does not precede a pointer operator, __based ignored.
- 0718 Variable in __based modifier must have pointer type.
- 0719 The type in a const_cast must be a pointer, reference, or pointer to member to an object type.
- 0720 A const_cast can only adjust type qualifiers; it cannot change the underlying type.
- 0721 Mutable is not allowed.
- 0722 ec_cannot_change_access [redeclaration of %n is not allowed to alter its access]
- 0723 Nonstandard format string conversion.
- 0724 Use of alternative token “<:” appears to be unintended.
- 0725 Use of alternative token “%%:” appears to be unintended.
- 0726 Namespace definition is not allowed.
- 0727 Name must be a namespace name.
- 0728 Namespace alias definition is not allowed.
- 0729 Namespace-qualified name is required.

- 0730 A namespace name is not allowed.
- 0731 Invalid combination of DLL attributes.
- 0732 `%n` is not a class template.
- 0733 Array with incomplete element type is nonstandard.
- 0734 Allocation operator may not be declared in a namespace.
- 0735 Deallocation operator may not be declared in a namespace.
- 0736 `%np1` conflicts with using-declaration of `%np2`.
- 0737 Using-declaration of `%npd1` conflicts with `%npd2`.
- 0738 Namespaces option can be used only when compiling C++.
- 0739 Using-declaration ignored; it refers to the current namespace.
- 0740 A class-qualified name is required.
- 0741 Argument types are: `%s`.
- 0742 Operand types are: `%s`.
- 0743 Using-declaration of `%n` ignored.
- 0744 `%n` has no actual member `%sq`.
- 0746 Incompatible memory attributes specified.
- 0747 Memory attribute ignored.
- 0748 Memory attribute may not be followed by a nested declarator.
- 0749 Memory attribute specified more than once.
- 0750 Calling convention specified more than once.
- 0751 A type qualifier is not allowed.
- 0752 `%npd1` was used before its template was declared.
- 0753 Static and nonstatic member functions with same parameter types cannot be overloaded.
- 0754 No prior declaration of `%np`.
- 0755 A template-id is not allowed.
- 0756 A class-qualified name is not allowed.
- 0757 `%n` may not be redeclared in the current scope]

- 0758 Qualified name is not allowed in namespace member declaration.
- 0759 `%n` is not a type name.
- 0760 Explicit instantiation is not allowed in the current scope.
- 0761 `%n` cannot be explicitly instantiated in the current scope.
- 0762 `%n` explicitly instantiated more than once.
- 0763 Typename may only be used within a template.
- 0764 `Special_subscript_cost` option can be used only when compiling C++.
- 0765 Typename option can be used only when compiling C++.
- 0766 Implicit typename option can be used only when compiling C++.
- 0767 Nonstandard character at start of object-like macro definition.
- 0768 Exception specification for virtual `%n1` is incompatible with that of overridden `%n2`.
- 0769 Conversion from pointer to smaller integer.
- 0770 Exception specification for implicitly declared virtual `%n1` is incompatible with that of overridden `%n2`.
- 0771 `%no1`, implicitly called from `%np2`, is ambiguous.
- 0772 Option `explicit` can be used only when compiling C++.
- 0773 `explicit` is not allowed.
- 0774 Declaration conflicts with `%sq` (reserved class name).
- 0775 Only “()” is allowed as initializer for array `%n`.
- 0776 `virtual` is not allowed in a function template declaration.
- 0777 Invalid anonymous union; class member template is not allowed.
- 0778 Template nesting depth does not match the previous declaration of `%n`.
- 0779 This declaration cannot have multiple “template <...>” clauses.

- 0780 Option to control the for-init scope can be used only when compiling C++.
- 0781 `%sq`, declared in for-loop initialization, may not be redeclared in this scope.
- 0782 Reference is to `%nd1`--under old for-init scoping rules it would have been `%nd2`.
- 0783 Option to control warnings on for-init differences can be used only when compiling C++.
- 0784 Definition of virtual `%n` is required here.
- 0785 Empty comment interpreted as token-pasting operator “##”.
- 0786 A storage class is not allowed in a friend declaration.
- 0787 Template parameter list for `%no` is not allowed in this declaration.
- 0788 `%n` is not a valid member class or function template.
- 0789 Not a valid member class or function template declaration.
- 0790 A template declaration containing a template parameter list may not be followed by an explicit specialization declaration.
- 0791 Explicit specialization of `%n1` must precede the first use of `%n2`.
- 0792 Explicit specialization is not allowed in the current scope.
- 0793 Partial specialization of `%n` is not allowed.
- 0794 `%nf` is not an entity that can be explicitly specialized.
- 0795 Explicit specialization of `%n` must precede its first use.
- 0796 Template parameter `%s` may not be used in an elaborated type specifier.
- 0797 Specializing `%n` requires “template<>” syntax.
- 0799 Nonstandard “asm” declaration is not supported inside a template.
- 0800 Option `old_specializations` can be used only when compiling C++.
- 0801 Specializing `%n` without “template<>” syntax is nonstandard.

- 0802 This declaration may not have extern “C” linkage.
- 0803 `%sq` is not a class or function template name.
- 0804 Specifying a default argument when redeclaring an unreferenced function template is nonstandard.
- 0805 Specifying a default argument when redeclaring an already referenced function template is not allowed.
- 0806 Cannot convert pointer to member of base class `%t2` to pointer to member of derived class `%t1`--base class is virtual.
- 0807 Exception specification is incompatible with that of `%nd%s= ' : ' .`
- 0808 Omission of exception specification is incompatible with `%nd`.
- 0809 The parse of this expression has changed between the point at which it appeared in the program and the point at which the expression was evaluated; `typename` may be required to resolve the ambiguity.
- 0810 Default-initialization of reference is not allowed.
- 0811 Uninitialized `%n` has a const member.
- 0812 Uninitialized base class `%t` has a const member.
- 0813 `const %n` requires an initializer--class `%t` has no explicitly declared default constructor.
- 0814 `const` object requires an initializer--class `%t` has no explicitly declared default constructor.
- 0815 Option `implicit_extern_c_type_conversion` can be used only when compiling C++.
- 0816 Strict ANSI mode is incompatible with long preserving rules.
- 0817 Type qualifier on return type is meaningless.
- 0818 In a function definition, a type qualifier on a `void` return type is not allowed.
- 0819 Static data member declaration is not allowed in this class.
- 0820 Template instantiation resulted in an invalid function declaration.

- 0821 “...” is not allowed]
- 0822 Option `extern_inline` can be used only when compiling C++.
- 0823 Extern inline `%n` was referenced but not defined.
- 0824 Invalid destructor name for type `%t`.
- 0825 Use of `%n` in a destructor call is nonstandard.
- 0826 Destructor reference is ambiguous--both `%n1` and `%n2` could be used.
- 0827 Virtual inline `%n` was never defined.
- 0828 `%n` was never referenced.
- 0829 Only one member of a union may be specified in a constructor initializer list.
- 0830 Support for `new[]` and `delete[]` is disabled.
- 0831 `double` used for `long double` in generated C code.
- 0832 `ec_no_corresponding_delete` [`%n` has no corresponding operator `delete%s` (to be called if an exception is thrown during initialization of an allocated object)]
- 0833 Support for placement delete is disabled.
- 0834 No appropriate operator delete is visible.
- 0835 Pointer or reference to incomplete type is not allowed.
- 0836 Invalid partial specialization--`%n` is already fully specialized.
- 0837 Incompatible exception specifications.
- 0838 Returning reference to local variable.
- 0839 Omission of explicit type is nonstandard (`int` assumed).
- 0840 More than one partial specialization matches the template argument list of `%n`.
- 0841 Ambiguous partial specialization `%no`.
- 0842 A template argument list is not allowed in a declaration of a primary template.

- 0843 Partial specializations may not have default template arguments.
- 0844 `%n1` is not used in template argument list of `%n2`.
- 0845 The type of partial specialization template parameter `%n` depends on another template parameter.
- 0846 The template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter.
- 0847 This partial specialization would have been used to instantiate `%n`.
- 0848 This partial specialization would have been made the instantiation of `%n` ambiguous.
- 0849 Expression must have integral or enum type.
- 0850 Expression must have arithmetic or enum type.
- 0851 Expression must have arithmetic, enum, or pointer type.
- 0852 Type of cast must be integral or enum.
- 0853 Type of cast must be arithmetic, enum, or pointer.
- 0854 Expression must be a pointer to a complete object type.
- 0855 A partial specialization of a member class template must be declared in the class of which it is a member.
- 0856 A partial specialization nontype argument must be the name of a nontype parameter or a constant.
- 0857 Return type is not identical to return type `%t` of overridden virtual function `%n`.
- 0858 Option `guiding_decls` can be used only when compiling C++.
- 0859 A partial specialization of a class template must be declared in the namespace of which it is a member.
- 0860 `%n` is a pure virtual function.
- 0861 Pure virtual `%n` has no override.
- 0862 `__declspec` attributes ignored.

- 0863 Invalid character in input line
- 0864 Function returns incomplete type %t.
- 0865 Effect of this `#pragma pack` directive is local to %n.
0866 %s is not a template.
- 0867 A friend declaration may not declare a partial specialization.
- 0868 Exception specification ignored.
- 0869 Declaration of `size_t` does not match the expected type %t.
- 0870 Space required between adjacent ">" delimiters of nested template argument lists (">>" is the right shift operator).
- 0871 Could not set locale %sq to allow processing of multibyte characters.
- 0872 Invalid multibyte character sequence.
- 0873 Template instantiation resulted in unexpected function type of %t1. (The meaning of a name may have changed since the template declaration--the type of the template is %t2.)
- 0874 Ambiguous guiding declaration--more than one function template %no matches type %t.
- 0875 Non-integral operation not allowed in nontype template argument.
- 0876 Option `embedded_c++` can be used only when compiling C++.
- 0877 Embedded C++ does not support templates.
- 0878 Embedded C++ does not support exception handling.
- 0879 Embedded C++ does not support namespaces.
- 0880 Embedded C++ does not support run-time type information.
- 0881 Embedded C++ does not support the new cast syntax.
- 0882 Embedded C++ does not support using-declarations.
- 0883 Embedded C++ does not support `mutable`.
- 0884 Embedded C++ does not support multiple or virtual inheritance.

- 0885 Invalid Microsoft version number.
- 0886 Pointer-to-member representation has already been set for %n.
- 0887 %t1 cannot be used to designate constructor for %t2.
- 0888 Invalid suffix on integral constant.
- 0889 Operand of `__uuidof` must have a class type for which `__declspec (uuid(...))` has been specified.
- 0890 Invalid GUID string in `__declspec (uuid(...))`.
- 0891 Option `vla` can be used only when compiling C.
- 0892 Variable length array with unspecified bound is not allowed.
- 0893 An explicit template argument list is not allowed on this declaration.
- 0894 An entity with linkage cannot have a variably modified type.
- 0895 A variable length array cannot have static storage duration.
- 0896 %n is not a template.
- 0897 Variable length array dimension (declared %p).
- 0898 Expected a template argument.
- 0899 Explicit function template argument lists are not supported yet in expression contexts.
- 0900 Nonmember operator requires a parameter with class or enum type.
- 0901 Option `enum_overloading` can be used only when compiling C++.
- 0902 Using-declaration of %n is not allowed.
- 0903 Invalid icode buffer size.
- 0904 Object pointed at not really `const`.
- 0905 Cannot use void.
- 0906 Cannot take address.
- 0907 Should be NULL.

0908 Pointer mismatch.

0909 Missing function prototype.

0910 Compiler tag validation.

0911 Symbol table underflow.

0912 Cannot complete type.

0913 Cannot determine file timestamp.

0914 %sq is incomplete.

0915 CFE generic error! %sq.

0916 Declaration mismatch.

0917 Label %sq is undefined.

0918 Possible unintended assignment in test.

0919 %sq has storage error.

0920 Struct/union inappropriate.

0921 Struct/union pointer mismatch.

0922 Struct/union type not allowed.

0923 Invalid digit.

0924 Static function %sq not defined.

0925 Unregistered sequence point.

0926 Constant expression or static address is required.

0927 Label %sq is unused.

0928 Identifier %sq must have struct or union type.

0929 Identifier %sq is not a field.

0930 _asm: escape uses more arguments than given.

0931 _asm: first parameter should be constant.

0932 _asm: __label() should not be in __obj_*() or __reg().

0933 _asm: __obj_*() should not be in __label().

0934 _asm: __reg_*() should not be in __label().

0935 `_asm: __call() should be in __reg_*().`

0936 `_asm: __frame_size() should be in __obj_*().`

0937 `ec_asm_bad_func [_asm: bad pseudo function %sq]`

0938 `_asm: bad expression in pseudo function.`

0939 `_asm: non-constant are disallowed in external _asm().`

0940 `_asm: bad field value in external _asm().`

0941 `Illegal/unknown target number.`

0942 `C++ style comments are not ANSI C compliant.`

I-Code Linker

adding already existing translation

Internal I-code linker error.*

adding duplicate symbol entry for “<symbol>”

The compiler encountered <symbol> twice as a global symbol. This may occur when two files being I-code linked contain a global symbol with the same name.

bad buffer size “<text>” — using <num>

The buffer size given for the -b option is 0 or not a decimal value. <num> is the I-code linker default number of bytes.

bad I-Code header sync code

The first four sync bytes of a potential I-code file are incorrect for an I-code file.

basic type collapse (<value>) in “<filename>”

Internal I-code linker error.*

can produce only one output file

The -o option can only be specified once on a command line.

can't find symbol in order array

Internal I-code linker error.*

*. If the message can be reproduced, try to isolate the problem and submit a [Product Discrepancy Report](#) to Microware.

can't get memory

The I-code linker is out of memory. This may be due to a large number of symbols or a very large function.

can't open '<filename>' name file

The file name given on the `-z` option is not accessible. Create the file or change the permission enabling the file may be opened for reading.

copy_initializer() can't do extended precision yet

Internal I-code linker error.*

copy_initializer() can't do long integers yet

Internal I-code linker error.*

debug information expected for 0x<value> in "<filename>"

Internal I-code linker error.*

debug information missing in "<filename>"

Internal I-code linker error.*

debug location 0x<value> not translated in "<filename>"

Internal I-code linker error.*

debug location <value> not translated in "<filename>"

Internal I-code linker error.*

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

duplicate declaration of symbol “<symbol>” in file “<filename>”

Internal I-code linker error.*

duplicate library symbol entry for “<symbol>” in “<filename>”

An I-code library contains two symbols with the same name.

file “<filename>” has invalid phase completion

The I-code file <filename> is not marked as a complete/valid I-code file.

flow block search failure for “<filename>”

Internal I-code linker error.*

flow block count mismatch for “<filename>”

Internal I-code linker error.*

flow block statement search failure for “<filename>”

Internal I-code linker error.*

host mismatch on “<filename>”

The host of an I-code file does not match that of previously encountered I-code files.

I-code buffers exhausted

The size of the i-code buffer needs to be increased with the `-b` option.

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

I-Code file format on “<filename>” too old/new

The I-code revision of <filename> is greater or less than the I-code optimizer anticipated. Recompiling the I-code file with the current compiler should correct the problem.

ilink: unknown option ‘<char>’

The compiler encountered the option -<char> and does not recognize it as a valid option.

illegal basic type <value> in file “<filename>”

Internal I-code linker error.*

illegal expression opcode (<value>) in “<filename>”

Internal I-code linker error.*

illegal initializer type (<value>) in “<filename>”

Internal I-code linker error.*

illegal statement opcode (<value>) in “<filename>”

Internal I-code linker error.*

illegal storage class (<value>) for “<filename>”

Internal I-code linker error.*

illegal type list ptr in basic type

Internal I-code linker error.*

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

incomplete lifetime information for “<symbol>” in “<filename>”

Internal I-code linker error.*

missing composite ptr in BT_COMP

Internal I-code linker error.*

missing debug ID entry in “<filename>”

Internal I-code linker error.*

no input

A file for linking was not identified to the I-code linker.

no libraries used in library construction

Libraries to link with cannot be specified when an I-code library is being built.

output file name required

An output file name was not specified. Use the `-o` option to specify an output file.

reference search failure for “<filename>”

Internal I-code linker error.*

target mismatch on “<filename>”

The target of an I-code file does not match that of previously encountered I-code files.

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

type mismatch for '<symbol>' in "<filename>"

The type on <symbol> in <filename> is different than the type for the same symbol encountered in a previous file.

unknown debug entry type (<value>) in "<filename>"

Internal I-code linker error.*

unknown debug location run type (<value>) in "<filename>"

Internal I-code linker error.*

warning: I-Code file byte order incorrect

The sync code in the I-code file looks as though the byte order on the host machine is different than that on the creator of the I-code file.

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

error: I-Code thread incompatibility detected. “<isect_name>” in “<filename>”.

This error (or warning, depending on the argument used with the option `-mt`), is generated when code marked as “non-thread-safe” is linked to code marked as “thread safe.” This may happen when threaded programs become linked to non-thread-safe versions of libraries.

To solve this problem, you can do one of two things:

- If the library you are using is your own and you know it to be thread-safe (even though it was not compiled with threading enabled), use `ilink`'s `-mts` option to create an inherently thread-safe library.
- If the library you are using is one other than your own, check for the presence of an `mt_ version` of the same library and link with that version. If there is no `mt_ version`, contact the supplier for more information on the threading capabilities of the library. Using `ilink`'s `-mtw` option can convert this error into a warning. Using `-mtq` can eliminate the warning altogether, but can also hide actual problems.

I-Code Optimizer

can't allocate memory

The I-code optimizer ran out of memory while processing the I-code file. Extremely long functions may cause the optimizer to consume large amounts of memory. If memory to allocate is unavailable, break large functions into smaller functions. A large number of functions in an I-code file may also cause the I-code optimizer to use more memory when performing function inlining but has no effect when function inlining is not being performed.

change of reference impossible for opcode 0x<hexvalue>

Internal I-code optimizer error.*

expression add to a non-expression statement

Internal I-code optimizer error.*

<filename> has improper sync code for an I-Code file

The I-code optimizer encountered an I-code file that does not begin with the proper sync bytes. Verify that the file in question is an I-code file.

<filename> is too old/new for this version of the optimizer

An I-code revision was encountered that does not match the format that the I-code optimizer expects. If the file is too old, recompile the source of the I-code file. If the file is too new, run the optimizer that was shipped with the rest of the compiler components in use.

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

<filename> not marked as a complete/valid I-Code file

The I-code optimizer encountered an I-code file marked as invalid or incomplete. Generally, this is because the file creator was abnormally interrupted before it finished writing the I-code file.

<function name>() found to contain no non-label statements

Internal I-code optimizer error.*

<function name>() to return value and returning no expression in <number> places

If return value checking has been enabled, this message may appear. The I-code optimizer encountered a function declared to return a value but the function has a point, or points, where it returns without a useful value. Change the source for the function to return a useful value at all times.

I-code buffers exhausted

The size of the i-code buffer needs to be increased with the `-b` option.

illegal factor for -k option

The `-k` option contains a syntactically incorrect argument. The argument to `-k` must be in the form:

`<space> : <time>`

`<space>` and `<time>` are strings of decimal integers or empty strings. If either field is left blank then the default value of 1 is used. If either value is zero then the other value is assumed to be the maximum value.

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

illegal size for -b option

The `-b` option was given a number it cannot parse. The argument to `-b` must be a string of decimal integers optionally followed by `k` or `K`.

illegal/unknown constant type 0x<hexvalue> found

Internal I-code optimizer error.*

improper statement opcode encountered (0x<hexvalue>)

Internal I-code optimizer error.*

label numbers wrapped, file is too big

The I-code optimizer exhausted unique label numbers available for the I-code file it is processing. The source file must be split into multiple smaller source files.

label(s) expected

Internal I-code optimizer error.*

no expression reference in supposed parent

Internal I-code optimizer error.*

no type for <function name>()

Internal I-code optimizer error.*

*. If the message can be reproduced, try to isolate the problem and submit a [Product Discrepancy Report](#) to Microware.

Note: <function name>() is NOT reducible, no loop optimizations done

The I-code optimizer encountered a function containing constructions that cannot be reduced. This can be caused by using `goto` statements to jump into the middle of a loop. Loop optimizations cannot be performed on functions that cannot be reduced. If loop optimizations are desired, restructure the function to use only structured programming techniques.

possible uninitialized variable '<symbol>' in <function name>()

If uninitialized variable warnings are enabled, this message may display. The I-code optimizer determined that `<symbol>` is referenced before it is given a value. The optimizer can be fooled into thinking this is the case. Correct code is generated in any case. For example, as the programmer, you might know that the loop goes around at least once in the following code segment, but the compiler cannot make that assumption. Therefore, it generates the message:

```
iopt: **** Possible uninitialized variable last in  
find_last() ****.
```

```
int find_last(void)  
{  
    for (i = 0; i < func(); i++)  
        last = i;  
    if (last == 10)  
        printf("Last is 10!\n");  
}
```

ran off function looking for a non-label statement

Internal I-code optimizer error.*

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

ran off leaf of expression tree/bad walk order error

Internal I-code optimizer error.*

reference to a composite not found

Internal I-code optimizer error.*

unknown symbol type encountered

Internal I-code optimizer error *

Back End

can't open file “%s”

Error opening specified file. Verify file permission.

can't open output file “<file>”

Error opening the file in which to write the generated code. Verify file or directory permissions.

duplicate symbols encountered

The back end determined that the same symbol name appears in more than one `psect` in the allocation of the final module.

error writing output file

An error occurred upon writing the output file. Several reasons exist for this error (for example, disk full or heavily fragmented). Reference the error code that the back end exits with for a possible error reason.

expected “-b[=<num>[k]”

An invalid option was passed to the back end. If the compiler executive invoked the back end and this error occurred, submit a Product Discrepancy report and the following information to Microware:

- The exact command line used to invoke the compiler executive
- Output written to standard error when the executive is invoked with the same command line used when the error occurred plus the option “-b”.

I-code buffers exhausted

The size of the i-code buffer needs to be increased with the `-b` option.

incomplete icode file

The I-code file that the back end is processing is invalid (for example, interruption of a `os9make` process).

internal error — emit_expr unknown op

An initializer contains an expression that the back end does not expect. Submit the following to Microware:

- **Product Discrepancy Report**
- I-code version of the smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — illegal dbg op

Invalid source debugging information was found in the input file. Submit the following to Microware:

- **Product Discrepancy Report**
- Smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — improper spill statement

Submit a **Product Discrepancy Report** to Microware.

internal error — initializer is not a string

An unexpected element was encountered where a string constant was expected. Submit the following to Microware:

- **Product Discrepancy Report**
- Smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — list_ident_refs not found

Submit a [Product Discrepancy Report](#) to Microware.

internal error — no pattern match on statement (<number>)

An I-code expression was encountered for which the back end cannot generate code (for example, incorrect spelling or use of object, register selection, or label generation pseudo functions in an `_asm()` statement). Submit the following to Microware:

- [Product Discrepancy Report](#)
- I-code version of the smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — no regs

Submit a [Product Discrepancy Report](#) to Microware.

internal error — NULL initializer string

An unexpected element was encountered where a string constant was expected. Submit the following to Microware:

- [Product Discrepancy Report](#)
- Smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — param spill

Submit a [Product Discrepancy Report](#) to Microware.

internal error — pattern not implemented

An I-code operator that cannot be processed was encountered. Submit the following to Microware:

- [Product Discrepancy Report](#)

- I-code version of the smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — simplify () no spill

Submit a [Product Discrepancy Report](#) to Microware.

internal error — sym not found

Expected information associated with a symbol cannot be found.
Submit the following to Microware:

- [Product Discrepancy Report](#)
- I-code version of the smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — translation overflow

The back end made several unsuccessful attempts to generate code for a particular function. Submit the following to Microware:

- **Product Discrepancy Report**
- I-code version of the smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — unknown expression opcode

An unfamiliar I- code statement type or operator was encountered. Submit the following to Microware:

- **Product Discrepancy Report**
- I-code version of the smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — unknown initializer type

An initialized variable with an initializer that the back end is unfamiliar with was encountered. Submit the following to Microware:

- **Product Discrepancy Report**
- I-code version of the smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — unknown statement opcode

An unfamiliar I-code statement type or operator was encountered. Submit the following to Microware:

- **Product Discrepancy Report**
- I- code version of the smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — wrong sym class for initializer string

An unexpected element was encountered where a string constant was expected. Submit the following to Microware:

- **Product Discrepancy Report**
- Smallest C source file you are able to create which produces this error message
- Command line used to compile the source file

internal error — memory allocation error

The back end requires more memory. Ensure that adequate memory is available for allocation to the back end. If the error persists, submit a **Product Discrepancy Report** to Microware.

no input file

The back end does not recognize an I-code file name for which to generate code. If the back end was directly invoked, verify that an input file name was correctly identified.

If the compiler executive invoked the back end and this error occurred, submit the following information to Microware:

- **Product Discrepancy Report**
- Command line used to invoke the compiler executive

- Output written to standard error when the executive is invoked with the same command line used when the error occurred plus the option “-b”.

not enough arguments for assembly-language escape (E)

An `_asm()` statement contains a format escape corresponding to an expression not in the expression list.

too many input files

The back end encountered more than one I-code file name for which to generate code. If the back end was directly invoked, verify that the command line does not contain more than one I-code file name, etc.

If the compiler executive invoked the back end and this error occurred, submit the following information to Microware:

- **Product Discrepancy Report**
- Command line used to invoke the compiler executive
- Output written to standard error when the executive is invoked with the same command line used when the error occurred plus the option “-b”.

undefined symbols encountered

The back end encountered an undefined symbol. Verify the symbol name and/or define the symbol.

unknown option ‘<option>’

An invalid option was passed to the back end. If the compiler executive invoked the back end and this error occurred, submit the following information to Microware:

- Command line used to invoke the compiler executive
- Output written to standard error when the executive is invoked with the same command line used when the error occurred plus the option “-b”.

Assembly Optimizer

assembler directives not allowed in replace text

Internal assembly code optimizer error. *

bad action in comparison list

Internal assembly code optimizer error. *

bad misc. action in hand optimization init

Internal assembly code optimizer error. *

bad mnemonic extension

A mnemonic has an extension that is not allowed or is not in the set of valid extensions for the processor.

can't find destination label on %s

Internal assembly code optimizer error. *

can't open <file>

The assembly language optimizer cannot access the necessary file. The file may not exist or you do not have permission to read it.

directive extension invalid

A directive has an extension that is not allowed.

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

error writing output file

A write error occurred while trying to write the optimized output. This may be because the device to hold the output is full.

find label error

Internal assembly code optimizer error. *

illegal save position used in replace text, '<char>'

Internal assembly code optimizer error. *

inlabel error

Internal assembly code optimizer error. *

invalid action in hand optimization init

Internal assembly code optimizer error. *

invalid text escape in replace line '<text>'

Internal assembly code optimizer error. *

match error, unknown macro in pattern

Internal assembly code optimizer error. *

no matching addressing mode for argument #<arg_num>

The assembly code optimizer encountered an instruction that has an argument with an improper form. If this is hand written assembly language, verify the syntax of the arguments.

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

out of memory

The assembly code optimizer ran out of memory while processing the assembly file.

removing too many instructions

Internal assembly code optimizer error. *

saving over a pointer in test

Internal assembly code optimizer error. *

syntax error in hand-optimization insert text

Internal assembly code optimizer error. *

syntax error in source file

The assembly code optimizer encountered a general syntax error in the source assembly language file. Refer to the line's text and positional indicator for the source of the syntax error.

unknown type passed to next_type

Internal assembly code optimizer error. *

unsupported macro in match2()

Internal assembly code optimizer error. *

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

Assembler

-j and -bt cannot be used together

The assembler was invoked using the incompatible options `-j` and `-bt`.

<processor> instruction

The indicated instruction is only legal for a `<processor>` type target. check the target option being used.

“endm” without “macro”

The assembler encountered an `endm` with no matching `macro`.

“endr” without “rept”

The assembler encountered an `endr` with no matching `rept`.

absolute addressing

An absolute addressing mode was used. This message is only a warning.

bad label

The statement's label has an illegal character or does not begin with a letter.

bad mnemonic

The assembler found a mnemonic in a mnemonic field that is not recognized or is not allowed in the current program section.

bad operand

An operand expression is missing or incorrectly formed.

bad option

An option is unrecognized or incorrectly specified.

bad psect name

An error was made when specifying the name field of a `psect` directive.

branch offset is non-even

The target of a branch is poorly aligned.

branch out of range

The target of the branch is too large to be used as the branch operand for the branch being used.

can't open file

The assembler encountered a problem while opening an input file.

can't open macro work file

The assembler encountered a problem while opening a macro work file.

comma expected

The assembler did not find a comma as expected.

conditional nesting error

The assembler found a mismatched `if/else/endif` conditional assembly directive.

constant definition

A constant definition is incorrectly formed.

constant overflow

The value of a constant expression exceeds the maximum signed 32-bit value.

constant value expected

A symbolic name was used in a context where only a constant value is allowed.

destination in <size> branch range

This warning indicates that a shorter, more optimal branch could be used in place of the branch instruction being used.

divide by zero

A divide by zero error was encountered during constant expression evaluation.

ERROR, can't open temp file <file>

An error occurred while attempting to open a temporary file. Possible causes include hardware failure, out of media error, or out of memory error.

error writing output file

An error occurred while trying to write to an output file.

expecting <char>

An error was encountered while parsing an instruction operand.

expression value must be pre-defined

A symbolic name is used in an expression prior to its being defined. The context in which the expression is used disallows such undefined symbolic use.

file close error

The assembler encountered a problem while closing an input file.

floating control reg expected

A floating point control register name was expected but not found.

floating register expected

A floating point register name was expected but not found.

illegal addressing mode

Use of the addressing mode in the instruction is disallowed.

illegal expression terminator

An error was encountered while parsing an instruction operand.

illegal external reference

Use of external names with assembler directives is disallowed. If an operand expression contains an external name, the only operation allowed in the expression is binary plus and minus.

illegal global symbol

An attempt was made to use a global label with the `set` directive.

illegal in <processor> mode

An attempt was made to use a segment register in an illegal mode.

illegal macro reference

A macro was used in a context where it is disallowed.

illegal register usage

A register name was found in a context where a register name is disallowed or invalid.

illegal size

An illegal size was specified for the given instruction.

illegal suffix

The assembler encountered an illegal suffix in an instruction.

illegal use of register designator

The register designator character was used in an illegal context.

incorrectly aligned expression

The expression value does not meet the alignment restrictions of the given context.

invalid <type> register

The register name given is illegal or of the incorrect type for the given context.

invalid register specification

The register name given is illegal or of the incorrect type for the given context.

invalid register type

The register name given is illegal or of the incorrect type for the given context.

label incompatible with 'spc' directive

An attempt was made to use a label with the `spc` directive.

label missing

This statement is missing the required label.

macro file error

The assembler encountered a problem while accessing the macro work file.

macro nesting too deep

Macro calls are nested too deeply. Macro calls can be nested up to eight levels deep.

mismatched operands

Incompatible register operands were specified for the instruction.

nested “macro”

A macro cannot be defined inside a `macro` definition.

nested “rept”

Repeat blocks cannot be nested.

no input file

An input file was not specified.

no matching encoding found

No encoding data was found for this instruction pattern.

no param for arg

A macro expansion is attempting to access an argument that was not passed by the macro call.

not a known register

An invalid register name was given.

offset truncated to 5 bits

A bitfield offset greater than 32 bits was given. The value is truncated to 5 bits.

only <reg> allowed

Something other than the given register name was used.

out of memory

A memory allocation request failed terminating the assembler processing.

parenthesis needed

An error was encountered while parsing an instruction operand.

phasing error

A label has a different value during the second pass than it did during the first pass. If other assembler messages do not precede this message, this message reflects a problem with the assembler.*

redefined label

The name occurs more than once in the label field other than on a `set` directive.

register designator '%' expected

The register designation character `%` is required prior to register name specification while in current context.

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

register expected

An invalid register name was given or some other expression was used in a context where a register operand was expected.

Spanreg must be specified for branch sizing

Branch sizing was enabled for the assembler, yet the `spanreg` directive was not used to set aside an assembler scratch register.

syntax error in floating point constant

An error was encountered while parsing a floating point constant expression.

terminator needed

An error was encountered while trying to parse an instruction operand.

too many args

Too many arguments were passed to the macro. A maximum of nine arguments may be passed to a macro.

too many object files

Only one `-o=` command line option is allowed.

too many operands

Too many operands were specified for the given instruction. Verify that there is a space between operands and comments appearing on the same line.

undefined org

Accessing the program counter (`'*'`) within a `vsect` is disallowed.

unexpected EOF

End of file was encountered prematurely.

unexpected end of line

The end of line was encountered prior to finding enough instruction/directive operands for the given instruction/directive.

unmatched quotes

The assembler could not find the expected beginning or ending quotation mark.

unresolved value

An unresolved symbolic value was used in a context where only constants or defined symbols are allowed.

value out of range

The value calculated for an expression is too large for the context in which the expression was used.

word sized immediate used with CCR

An attempt was made to use a word sized immediate value in an instruction that only allows 1-byte immediate values, to be combined with the CCR register.

Prelinker

error: executing <command-line>

Prelinker failed executing `xcc`.

error: unrecognized option: <option>

Illegal option on prelinker command line.

error: out of memory

Could not allocate memory.

error: invalid input format

Invalid output generated by `libgen -ln`.

error: bad instantiation information file

Instantiation assigned to more than one file.

error: command line error

Bad prelinker command line.

error: instantiation loop

Too many prelinker iterations.

error: library <name> does not exist in the specified library directories

warning: an error occurred during name decoding of <name>

Mangled C++ name could not be demangled.

error: cannot open object file name list file <name>

For `-N` option; issued when list file cannot be created.

error: cannot create instantiation information file <name>

File or directory may be read only or media may be full.

error: cannot change to directory <name>**warning: no output produced by nm**

Possible configuration problem; usually means `libgen -ln` failed for some reason.

error: unable to create process for nm command

OS-9 pipe or fork has failed.

error: <name> has been referenced as both an explicit specialization and a generated instantiation**error: file <name> is read-only**

Cannot write to file.

warning: nm returned a nonzero error status

Some ROF(s) may be corrupted or non-existent.

Object Code Linker

-a or -j valid only when specifying OS-9/68K target

The options `-a` and `-j` may only be used when targeting OS9/68K target. Refer to the object code linker `-t` option for target codes.

can't allocate enough memory

The linker cannot obtain enough memory to do the linkage. Memory use requirements depend on many factors: number of input files, number of `psect`s, number of global symbols, and undefined references. The largest use of memory is during the second pass when each `psect`'s references must be adjusted for the final program module. To perform this, the linker must be able to get as much memory as the largest sect used. A `psect` that is 128K long requires a 128K buffer to link.

can't open debug file

The linker could not open the `.dbg` file associated with the output module.

can't open file

The linker could not open a specified input file. Possible causes are lack of access permissions, a non-existent file, or unavailable free memory.

can't set file attributes for file <file>

The linker could not set file attributes for the specified file. Possible causes are lack of access permissions, a non-existent file, or unavailable free memory.

common block <common> in psect <psect1> has type conflict with symbol in psect <psect2>

The linker found the same common block symbol in multiple `psects` with conflicting type definitions.

duplicate symbols encountered

The linker determined that the same symbol name appears in more than one `psect` in the allocation of the final module. Consider the case of a program with multiple source files. If more than one `psect` defined the global data symbol `count` the following error message would be issued:

```
error - symbol name 'count',  
defined by psect 'cmd.c',  
file 'cmd.r' already defined by psect 'main.c', file 'main.r'
```

error reading file

The linker cannot read the file. Either a physical error occurred or the input file was incorrectly formatted. All input files must be output from the assembler.

error seeking in file

The linker cannot seek in a file. This could result from a physical error on the disk device.

error specifying alignment boundary

An invalid alignment was given to the `-x` or `-b` option on the command line.

error specifying library name

An invalid library file name was given on the command line. Check the operands used with the `-l` option.

error specifying memory size

An invalid operand was given to the `-m` option.

error specifying target

An invalid target processor/operating system was given. Check the operand used with the `-t` option.

error writing file

The linker could not write the file. Possible causes are disk errors or media full.

errors encountered

This message should be accompanied by messages specific to the errors encountered.

file <file> contains a 6809 module

The linker encountered a module from the 6809 assembler.

file <file> is not a relocatable module

The ROF module header in `<file>` was either absent or incorrectly formed. All ROF object headers start with the bytes: `$DE $AD $FA $CE`. Use the `dump` utility on the input file to verify this. Giving the wrong file on the command line is the most likely cause.

Illegal relative reference to symbol <symbol> in the <type> area

A relative reference has been encountered in a context where such a reference is illegal.

Illegal use of reference in binary operand

An attempt was made to use a non-relative reference to a code or data item in an illegal context.

Illegal use of reference in negative operand

An attempt was made to use a non-relative reference to a code or data item in an illegal context.

Illegal use of reference in unary operand

An attempt was made to use a non-relative reference to a code or data item in an illegal context.

initialized data (or jumtable) allowed only on program or trap handler modules

For OS-9 for 68K, initialized data is supported only for program modules (entered by `F$Fork`) or trap handler modules (entered by `F$TLink`). Modules such as system modules, device drivers, and file managers cannot have initialized data. The C Compiler generates initialized data when C initializers appear for static or global data. The assembler generates initialized data when a data initialization directive (for example, `dc` or `dz`) appears in a `vsect`.

If the `-i` option is specified, the linker generates a program style module header for system modules and allows system modules initialized data. This does not imply that the resulting module is recognized by the 68K kernel.

internal error reference location error

Internal linker error. * This is caused when the linker receives unexpected information from the assembler. If this happens, be sure the assembler and linker are properly installed on the system from the original distribution medium.

internal error-unknown reference type

Internal linker error.*

jmp total guess in <psect> referencing <symbol> in <psect>

Internal linker error.*

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

no data storage allocation (vsect) allowed on non-object modules

Only modules of type `object code` can contain data storage allocation. Other module types (usually language runtime interpreters) define data storage allocations in a different manner.

no root psect found

The linker did not find a `psect` with a non-zero type/language field. The first `psect` encountered should be the root `psect` and have this field defined.

non-remote data allocation exceeds <bytes> bytes

The amount of non-remote memory used by the linked ROFs exceeds the maximum allowed for this processor. If compiling a C or C++ file, re-compile with the correct sub-options. (Refer to the appropriate processor-specific Executive and Phase Information for a description of sub-options.) If compiling an assembly code file, use remote `vsect` to reduce the amount of non-remote allocation.

odd count for crc

Internal linker error.*

psect <psect> in file <file> contains assembly errors

The linker encountered a module with assembly errors. Fix the errors and relink.

psect <psect> in file <file> created by assembler too new for this linker

The assembler and linker programs are not compatible editions. Be sure the correct programs are installed in the execution directory.

*. If the message can be reproduced, try to isolate the problem and submit a **Product Discrepancy Report** to Microware.

psect <psect> in file <file> has illegal rof edition. Must be re-assembled.

The ROF edition number in the given file is either illegal or obsolete. Re-assemble the source code to create an up-to-date ROF.

psect <psect> in file <file> is replacing psect <psect> in library <library>

This is a warning message indicating that all symbols defined in one `psect` were also found in a later `psect` from the named library. The first `psect` is used by the linker.

psect <psect> in file <file> rof<4 and code>32k. Must be re-assembled.

This is caused when the linker processes an old version of assembler output that contains more than 32K of code. Reassemble the source file to fix this problem.

root psect found in both <file1> and <file2>

Only one root `psect` is allowed for a program. A root `psect` is defined as a `psect` in which the Type/Language field is non-zero. The root `psect` is the initial `psect` from which all external references are resolved.

signal raised, abnormal termination error

The linker received signal.

signal raised, floating point error termination

The linker received signal.

signal raised, illegal instruction termination

The linker received signal.

signal raised, illegal memory reference termination

The linker received signal.

signal raised, termination request sent to program

The linker received signal.

signal raised, user termination

The linker received signal.

symbol <symbol> must also be defined to replace psect <psect>

This error indicates that one or more symbols from <psect> was defined by prior psects (possibly inadvertently), but not all the symbols from <psect> were defined. To replace a psect it is necessary to redefine all the global symbols of the psect. The following scenario illustrates the cause of this error message:

- a psect in a library defines the code symbol `alloc` and the data symbol `alloc_count`
- a user writes a function called `alloc` and references the data symbol `alloc_count`
- the linker registers the definition of `alloc` and the reference to `alloc_count`
- the linker emits this error message when it attempts to pull in the psect containing both `alloc` and `alloc_count` from the library
error - symbol 'alloc_count'
must also be defined to replace psect 'alib.c'.

symbol <symbol> unresolved, referenced by <psect>

The linker has encountered a reference to a symbol that has no definition.

**The value <value> is too large for <n>-bit <type> field [Caused by reference to symbol '<sym>'.]
 The offending expression is represented by tree #<num> of psect '<psect>' in file '<file>'.
 The expression is referenced at offset <offset> in the <code or data> area.**

The linker calculated a value for a reference that is too large to fit in the bit field of that reference. This is usually due to insufficiently long data or code area references. Add processor specific sub-options to lengthen code or data area references. Generally, errors like this result from insufficiently long code (.lc) or code area data (.lcd) references.

If the second line appears in the error message, determine the nature of the symbol to know whether it's a data area, code area data, or code reference. Enlarge the references to the appropriate area for the specified ROF to successfully link the module.

If the second line does not appear, then recompile the specified file again, stopping after the assembler, and request a listing from the assembler (-asglx). Examine the listing at the specified offset to determine what type of reference caused the problem.

Following these steps ensures that only the references that need to be made long are made long. I-code linking with data area layout automatically generates the correct references for all code and data items.

The following is an example error message, generated while compiling for PowerPC:

```
linker: error - The value $10000 is too large for 16-bit signed field.
The offending expression is represented by tree #0
of psect `main.c` in file `main.r`.
The expression is referenced at offset $2 in the code area.
```

Generating a listing for main.c generates this output:

```
00001 * UCCaoPPC
00002                psect    main.c,0,0,0,0,0
00003      00000000    align
00004                main:
00005 0000=80620000    lwz     r3,ext_symbol(r2)
00006      00000004    ends
```

From this we can see that offset \$2 (0x2) refers to the lower 16-bits of the `lwz` instruction. The base register used is `r2 (gp)`, thus it's a data area reference. This indicates that `,ld` should be added to the compile command to force all references to data to be 32-bits instead of 16-bits. Then, \$10000 will be a valid value for `ext_symbol`.

The value <value> is too large for use with the <op> operator

The linker calculated a value for a reference which is too large to be a valid operand for the given operator.

The value of symbol <symbol> is poorly aligned

The linker calculated a value for a reference which does not meet the alignment criterion of that reference.

Unknown operator encountered

The linker encountered an operator in an expression tree which is undefined or unrecognized by the linker.

Unknown operator for this processor

The linker encountered an expression operator that is unsupported on this processor. Check the input ROF target versus the argument used with the `-t` option.

unknown option -<char>

The linker does not recognize the given option.

Warning! '`_m_init`' and '`_m_term`' expected for trap handler modules

The `_m_init` and `_m_term` symbols should be defined in the code to define the trap handlers initialization and termination entry points. Old style trap handlers did not use this mechanism, so this warning is seen when their code is re-linked.

Warning! ‘_m_share’ expected for Drvr, Fman, and Sys module types

The symbol `_m_share` should be defined in the code to point to shared data to be accessed by the kernel, file manager, or other external code. This warning is printed for such modules whenever this field is not set.

Warning! ‘_syscmnt’ no longer supported

This message is displayed for code that defines the symbol `_syscmnt`. The linker no longer treats this symbol in any special way.

error: Thread-incompatibility detected in psect: ‘<psect_name>’ in file ‘<filename>’ (psect’s level is <non->thread-using, <non->thread-safe; prior were <non->thread-using, <non->thread-safe)

error: Thread-incompatibility detected in library file: ‘<filename>’ (library’s level is <non->thread-using, <non->thread-safe; prior psects were <non->thread-using, <non->thread-safe)

This error (or warning, depending on the argument used with the option `-mt`), is generated when code marked as “non-thread-safe” is linked to code marked as “thread safe.” This may happen when threaded programs become linked to non-thread-safe versions of other psects or libraries.

To solve this problem, you can do one of two things:

- If the file you are using is your own and you know it to be thread-safe (even though it was not compiled with threading enabled), use the `-mts` option on either `libgen` or the assembler to create an inherently thread-safe file.

- If the library you are using is one other than your own, check for the presence of an `mt_ version` of the same file and link with that version. If there is no `mt_ version`, contact the supplier for more information on the threading capabilities of the file. Using the linker's `-mtw` option can convert this error into a warning. Using `-mtq` can eliminate the warning altogether, but can also hide actual problems.

Appendix B: Migrating to Ultra C++ version 2.1

If you are moving from Ultra C++ version 2.0 to version 2.1, you need to be aware of the following changes:

- **Library changes:** The C++ libraries accompanying the compiler are now available in two versions, one with C++ exception support enabled and one without. This allows C++ code not using exceptions to compile to smaller and faster machine code. See the section [Compiling with Exceptions Disabled](#) in Chapter 12.
- **International character set support:** Ultra C/C++ now accepts Japanese Kanji and European non-ASCII characters in strings, character literals and comments. (This may need support from the host system).
- **enum is a true type:** It is distinguished from `int`, and overloading can be done using `enums` alone.

The rest of this chapter is primarily intended for those migrating from version 1.3 of Ultra C++ to version 2.1. The following sections are included:

- [New Language Features](#)
- [Language Features in the C++ Standard but Not Accepted](#)
- [Headers](#)
- [Namespaces](#)
- [Operators new and delete](#)
- [New Template Features](#)
- [The Standard C++ Library](#)
- [Prelinker](#)





Note

Ultra C++ version 2.1 supports a large subset of the ANSI C++ features. Therefore, code written using Ultra C++ version 1.3, which was largely an implementation of C++ as defined in ***Ellis & Stroustrup: The Annotated C++ Reference Manual (ARM)***, may need modification. Much of the code should compile unchanged, as long as it does not use the newer language features like templates, exceptions, and the new keywords. However, even with the older part of the language, new errors may be encountered where previously there were none. This is the result of certain changes in semantics of the language in the newer ANSI definition. Ultra C++ conforms very closely to the December 1996 Draft of the Standard for both the base language and the library.

New Language Features

1. Control falling off the end of `main` is treated as returning 0.
2. The precedence of the third operand of `?` is changed to conform to the ANSI C++ Standard.
3. The dependent statements of `if`, `while`, `do-while` and `for` are considered scopes. These statements can cause problems with older code. The following code example no longer compiles:

```
void f()
{
    for (int i = 0; i < 10000; ++i)
        ;
    i = 1; // error: "i" is undefined
}
```

The code does not compile because the scope of the variable `i` declared in the `for` statement is restricted only to the end of the statement. Therefore, the assignment following the `for` statement is illegal. Compiling code in `cfront` compatibility mode gets rid of the problem, but this solution is not a recommended.

4. Parameters of type pointers to arrays of unknown bounds are flagged as errors.
5. Given a class `T`, the notation `T()` declares a temporary of type `T` and initializes it using the default constructor, if defined. If the class `T` has no non-trivial constructor, the object gets initialized to zero, as would a static object of the same class. `T` can be a built-in type or a user-defined type.
6. A cast can be used to select one out of a set of overloaded functions when taking the address of a function. The following code is an example.

```
void f(int); // (1)
void f(void*); // (2)
void *p = (void*)(void*) f; // address of (2)
```

7. A reference to `const volatile` cannot be bound to an r-value.

8. Qualification conversions such as from `T**` to `T const* const*` are allowed. Example code follows.

```
void f(const char *const *p)
{
    // ...
}
int main(int argc, char **argv)
{
    f(argv); // ok with ANSI C++
}
```

9. Digraphs are recognized.

10. The following operator keywords are recognized:

<code>and</code>	<code>or</code>	<code>not</code>	<code>and_eq</code>
<code>or_eq</code>	<code>not_eq</code>	<code>bitand</code>	<code>bitor</code>
<code>compl</code>	<code>xor</code>	<code>xor_eq</code>	

11. Static data declarations can be used to declare member constants. An example is:

```
class limits {
    static const int max = 0x7FFFFFFF;
    static const int min = 0x80000000;
};
```

This is useful in writing code which does compile time computation as in the following example.

```
template<int N> struct pow2 {
    static const int value = pow2<N-1>::value * 2;
}
template<> struct pow2<0> {
    static const int value = 1;
}
//...
int i = pow2<16>::value; // calculated at compile time
```

The compiler can optimize the space required for objects declared `static const`. Often, no space is allocated at all, especially for scalar types, unless the address of a static `const` object is taken.

12. `wchar_t` is a distinct type and a keyword.

13. `bool` is a distinct type and keyword; it can be assigned the values `true` and `false` (which are also keywords). All relational expressions generate results of type `bool` now. However, standard conversions from `bool` to `int` are defined, so integral expressions can be mixed with boolean ones as before.
14. Run time type identification (RTTI) is implemented via the `dynamic_cast` and `typeid` operators.
15. Array `new` and `delete` are implemented. These can be redefined by the user if required. Previously in Ultra C++ 1.3, only one version of `new` and `delete` operators sufficed for both single object and array allocations and deallocations. This is no longer true. Instead, code that defines allocation and deallocation operators for classes now needs to define the `array-new` and `array-delete` versions also if array allocation or deallocation is being used. If this is not done, the global `array-new` and `array-delete` operators are used, which is probably not what you want.
16. New style casts, `static_cast`, `reinterpret_cast`, `const_cast` and `dynamic_cast` are implemented.
17. Definition of a nested class outside the enclosing class is allowed.
18. `mutable` is accepted on non-static data members. For example:

```
class C {
public:
    mutable int i;
    int j;
};
const C x;
// . . .
x.i++; // allowed
x.j++; // error
```
19. Namespaces are implemented, including `using` declarations and directives. The standard headers define all names in the namespace `std` with the exception of the new style C headers such as `<cstdio>` and `<cstddef>`. (This does not conform to the ANSI Standard and will be addressed in a future release.)

20. `explicit` is accepted to declare non-converting constructors. An example is:

```
class C {
public:
    explicit C(int i);
    C(const C&);
};

C x = 1; // can't use C(int) for conversion
```

21. The scope of the variable declared in the initialization part of a `for` statement extends only to the end of the statement. For example,

```
for (int i; i < 10; ++i)
;
i = 0; // not the same i as in the loop above
```

22. The distinction between trivial and non-trivial constructors is implemented. A constructor is trivial if and only if

- It is an implicitly declared default constructor
- Its class has no virtual functions or virtual base classes
- All direct base classes of its class have trivial constructors
- For each non-static data member of its class, a trivial constructor exists for the member's class

23. The linkage specification is now part of a function's type. This means that pointers to C functions have a different type than pointers to C++ functions. However, overloading cannot be done on this basis. Casting can be used to get around this difference when passing pointers to C functions as arguments to C++ functions. For example:

```
void F(void (*f)(void));
extern "C" f(void);

F(f); // error
F((void (*)(void))f); // ok
```

When passing pointers to C++ functions as arguments to C functions in a C++ program, the same problem arises. See the next example.

```
extern "C" F(void*)(int);
void g(int);
```

```
F(g); // error: g does not match F's parameter type
```

The solution is to declare a `typedef` and use casting as in the following example:

```
extern "C" typedef void (*CFPtr)(int);
F(CFPtr(g)); // ok after casting
```

Another solution is to use the new style casts:

```
F(reinterpret_cast<CFPtr>(g)); // ok after casting
```

This casting technique can also be used when using C function pointers whose arguments are left unchecked. For example,

```
void install(void(*f)())
```

in C means that no checking is done for arguments of functions passed to `install()`. Any function returning `void` can be passed as an argument. Even after adding an `extern "C"`, in C++ the prototype means that `f()` is a C function taking zero arguments, which is not the same as the C declaration. Casting can be used to break the strong type checking in this case. Since Ultra C++ uses the same ABI for both C and C++ functions, using function pointers after casting does not cause errors.

24. `extern inline` functions are supported; the default linkage for inline functions is external. However, this does not imply separate compilation of inline functions: the "extern" refers only to the linkage specification, and the definition of an inline function must still be present in any translation unit that references it; this is a language requirement. Inlining in Ultra C++ is controlled by the optimization level. For space optimization, inlining is usually suppressed.
25. A `typedefed` name may be used in an explicit destructor call.
26. New template features, like new specialization and explicit instantiation syntax, member templates and default template arguments, have been implemented.

Language Features in the C++ Standard but Not Accepted

1. Virtual functions differing only by their return types from the overridden functions in the base classes (covariance) are not implemented. For this reason, the following does not compile:

```
class B {
public:
    virtual B& f();
};
class D : public B {
public:
    D& f();
};
```

2. enum types cannot contain values larger than an int.
3. reinterpret_cast does not allow casting a pointer to member of one class to a pointer to a member of another class unless the classes are related.
4. Explicit qualification of template functions is not implemented. Consequently, the template parameters must be the types of at least one of the function parameters. For example, the below code generates an error, even though it is correct according to the Standard.

```
template<class T> T f() { /* ... */ }
f<int>();
```

5. In a reference of the form `f()->g()`, with `g()` a static member function, `f()` is not evaluated. This requires that `f()` be evaluated.
6. Partial specialization is not implemented. This will be addressed in future releases.
7. Placement `delete`, regular or array version, is not implemented. This will be addressed in a future release.
8. Template parameters are not implemented.
9. Putting a `try/catch` around the initializers and body of a constructor is not implemented.

10. String literals do not have `const` type. The Standard requires them to have type "array `n` of `const char`"; currently these have the type "array `n` of `char`".
11. Universal character set escapes (such as `\uabcd`) are not implemented.

Headers

The standard headers no longer have the `.h` extension. For example, `<iostream.h>` is now simply `<iostream>`. The list of C++ standard headers (those specific to C++ only) includes:

<code><algorithm></code>	<code><bitset></code>	<code><complex></code>	<code><deque></code>
<code><exception></code>	<code><fstream></code>	<code><functional></code>	<code><iomanip></code>
<code><ios></code>	<code><iosfwd></code>	<code><iostream></code>	<code><istream></code>
<code><iterator></code>	<code><limits></code>	<code><list></code>	<code><locale></code>
<code><map></code>	<code><memory></code>	<code><new></code>	<code><numeric></code>
<code><ostream></code>	<code><queue></code>	<code><set></code>	<code><sstream></code>
<code><stack></code>	<code><stdexcept></code>	<code><streambuf></code>	<code><string></code>
<code><typeinfo></code>	<code><utility></code>	<code><valarray></code>	<code><vector></code>

Ultra C++ maps the include file names to exactly what is inside the `< >` unlike some compilers which map these names to files with `.h` or some other extension. Therefore, if you have old headers like `<iostream.h>` in your code, remove them or move them to where the compiler cannot include them in your sources.

C specific headers are available through a new naming scheme: the `.h` extension is dropped and a `c` is prefixed to a header name. For example, `<stdio.h>` becomes `<cstdio>`. These headers are still available under their original names. The header `<strstream>` is provided for backward compatibility; however it is a deprecated feature of the Standard and may vanish from future versions. The header `<sstream>` provides a more modern version of in-memory streams.

Namespaces

All C++ Standard library-defined names are now in the namespace `std`, which is itself contained in the global namespace. The global namespace can be referred to by the empty name; therefore, `::std` is the correct form of the Standard Library namespace. Fully qualified names or `using` directives are required to access names in the Standard Library. For example, the `hello, world` program, previously written as follows:

```
#include <iostream.h>
int main()
{
    cout << "hello, world" << endl; return 0;
}
```

The `hello, world` program can now be written either as:

```
#include <iostream>
int main()
{
    using namespace std; // can also be at file scope
    cout << "hello, world" << endl;
    return 0;
}
```

or as:

```
#include <iostream>
int main()
{
    std::cout << "hello, world" << std::endl;
    return 0;
}
```

All C data names need to have their linkage specified if declared within a namespace. For example,

```
namespace OS {
    extern "C" void* kmem_start;
}
```

Referring to `kmem_start` always generates a reference to the unmangled C name. If the "C" linkage specifier is absent, linkage defaults to C++ and the name is mangled accordingly to reflect that it belongs to namespace `OS`. Even when external C data is not declared within a namespace, it is still a good idea to specify the linkage as "C" especially when the declaration is in a header and might be included inside a namespace.

The namespace names `__unix` and `__mw` are reserved in this implementation. In addition, all identifiers beginning with an underscore should be treated as reserved; user code should avoid declaring such names.

Operators new and delete

`new` and `delete` now come in 10 predefined flavors declared in header `<new>`, with an arbitrary number of user defined extensions possible. The predefined operators are:

1. `void* operator new(size_t) throw(std::bad_alloc);`
2. `void* operator new(size_t, const std::nothrow_t&) throw();`
3. `void* operator new[](size_t) throw(std::bad_alloc);`
4. `void* operator new[](size_t, const std::nothrow_t&) throw();`
5. `void* operator new(size_t, void*) throw();`
6. `void* operator new[](size_t, void*) throw();`
7. `void delete(void*) throw();`
8. `void delete(void*, const std::nothrow_t&) throw();`
9. `void delete[](void*) throw();`
10. `void delete[](void*, const std::nothrow_t&) throw();`

All of these signatures are in the global namespace; therefore, no using directive or namespace qualification is necessary. Nos.1 and 7 will be familiar as the "old" versions of `new` and `delete`. However, `new` now throws a `std::bad_alloc` exception when no more memory is available and no user-defined `new_handler` had been set. If the older behavior is desired where a null pointer is returned, version No. 2 can be used. The second argument (`const std::nothrow_t&`) is a dummy argument to `new` meant to call the non-throwing version. It is called as

```
Object* pointer;
pointer = new (std::nothrow) Object; // call No. 2
```

or

```
pointer = new (std::nothrow) Object[100]; // call No. 4
if (pointer == 0)
    // new failed
```

`std::nothrow` is a predefined constant in the Standard Library (declared in header `<new>`) of type `std::nothrow_t`.

A new concept introduced into the ANSI C++ proposal is the placement `new` and `delete` operators (versions 5, 6). Here the operator simply returns the second argument which allows the user to construct an object at some arbitrary (and presumably properly aligned) location. For example:

```
Object object_memory_area[10]; // aligned properly
Object* volatile object_ptr;
object_ptr = new(&object_memory_area[0]) Object;
```

Placement `delete` is not yet implemented but will be in future versions of Ultra C++. Placement versions do not throw any exceptions and the responsibility for managing memory is on the user.

All of the predefined `new` and `delete` operators can be redefined by the user, and additional parameters can also be defined. Redefinition means providing an alternate definition for the above operators than that supplied by the C++ Standard (runtime) Library. Any other operators with differing signatures are treated as overloads and do not conflict with the predefined ones. Since exception specifications do not form part of C++ function signatures, i.e., overloading cannot be done using exception specifications alone, redefining any of the predefined operators must have the exact same exception specifications. `new` and `delete` operators may also be defined for a class, in which case these are called when the corresponding `new` and `delete` expressions are used.

New Template Features

While template features have been enhanced to correspond more closely to that specified in the Standard, some features which push the limits of compiler technology are not yet implemented. The following enhancements are now available:

1. Type template parameters are permitted to have default arguments.
2. Template friend declarations and definitions are permitted in class definitions and class template definitions.
3. Function templates may have non-type template parameters.
4. The new specialization syntax is implemented. All specializations must now be specified using this syntax. For example:

```
template<class T>
inline void copy(T* dest, const T* source, int N)
{ while (N-- > 0) *dest++ = *source++; }

template<> // specialization for T = char
inline void copy(char* dest, const char* source, int N)
{ memcpy(dest, source, N); }
```

5. The new explicit instantiation syntax is implemented. All explicit instantiations must now use this syntax. For example:

```
template<class T> class Stack { /* ... */ };
// explicitly instantiate for T = int
template class Stack<int>;
```

6. Instantiation pragmas are still available; however, their use is now deprecated.

7. The `typename` keyword is recognized (and needed) to specify certain identifiers defined in class templates as denoting type names when it would be difficult or error prone for the compiler to deduce the same. For example,

```
template<class Types> class container {
    typename Types::int_type  int_data;
    typename Types::ptr_type  ptr_data;

    void f();
    /* ... */
};

class system_types {
public:
    typedef short int_type;
    typedef char* ptr_type;
};

container<system_types>    x;

template<class Types>
void container<Types>::f()
{
    // "typename" needed below!
    Types::int_type *int_ptr; // decl or expression? error!
}
```

Then `x.int_data` is of type `short` and `x.ptr_data` is of type `char*`. In member `f()`, the compiler needs `typename` to decide whether the statement is a (pointer) declaration or a multiplication expression. Note: use of `typename` is only allowed/needed within template definitions.

The Standard C++ Library

The new C++ Standard Library uses templates heavily. As a result, compilations can be slower. The advantage is that no runtime penalties occur, such as in the case of libraries which use inheritance to simulate being generic. Also, there is greater flexibility in the number of types that the library provides. For example, where previously there were only class `complex` available with `double` real and imaginary components, we can now have `complex<double>`, `complex<float>`, and so on. A template-based container and algorithm library is provided which provides generic and efficient implementations of lists, queues, stacks, sets, maps, and so forth, along with generic algorithms on which to operate. Algorithms can usually work with a diverse range of container classes by using the various iterator types provided by the container classes.

The I/O part of the Standard Library is also template based, mainly to accommodate both `char` and `wide-char` types. This is mostly transparent to a user of the traditional `iostreams`. Since the class names have changed in the newer library (but old names are retained via `typedefs`), forward declaring class names such as `istream` and `ostream` does not work. This is because the actual class names are:

```
template<class charT, class Traits> class basic_ostream;  
template<class charT, class Traits> class basic_istream;
```

with `istream` and `ostream` being `typedefs` of instantiations of these classes for `charT=char` and `Traits=char_traits<char>`. For `iostreams` forward declarations, the header `<iosfwd>` is defined which contains forward declarations for all the classes in the `iostreams` hierarchy.

The Standard Library now reports errors by throwing exceptions. All exceptions types used by the Library are based on the root class `std::exception` defined in the header `<exception>`. The simplest way to catch an exception thrown by the Library is to set up an exception handler for objects of type `const std::exception&` and use the virtual function `what()` to determine what kind of an exception was thrown (`typeid` can be used too).

Example

```
#include <iostream>
#include <exception>

using namespace std;

int main()
{
    try {
        // use iostreams here
    }
    catch (const exception& e) {
        clog << e.what() << endl;
    }
}
```

Using a `const` reference for the caught object type allows the virtual member function `what()` to report the type of the thrown object correctly. If a reference is not used, `what()` always binds to `std::exception::what()`.

For finer control over exception handling, use the classes defined in the header `<stdexcept>` or the exception class `std::ios_base::failure` which is used by iostreams to report failure.

While the Standard Library accompanying Ultra C/C++ 2.1 is close to what the December 1996 (Kona) specifies, some differences need mentioning:

- In the classes defined in `<stdexcept>` and the class `std::ios_base::failure`, there is no constructor taking a `std::string` argument. Instead, a constructor taking `const char*` is supplied. Thus, the statement

```
throw std::runtime_error(string("error"));
```

won't compile, but the statement

```
throw std::runtime_error("error");
```

compiles and executes with the same effect. If string operations are desired, such as concatenation, etc., then the `data()` member function of class `string` can be used, e.g.,

```
throw std::runtime_error((string("error ") +
```

```
int_to_string(code)).data());
```

- Since template functions templated by return type alone are not supported, code using the facet classes in `<locale>` needs to use workarounds.
- Placement `delete` and placement `array-delete` are not implemented. These will be implemented in a future compiler release.
- The new style C headers like `<cstdio>`, `<cstdlib>`, `<cstddef>`, etc. do not enclose their contents in `namespace ::std` as required by the standard. To achieve that effect we can use

```
namespace std {
#include <cstdio>
} /* . . . */

std::printf("hello, world\n");
```

However, due to some naming conflicts in the current implementation, this approach is not always guaranteed to work. For now, it is simpler to use the old style C headers like `<stdio.h>`, etc., although in some cases, like `<cmath>`, it may be preferable to use the newer style headers because of additional functionality available. Future releases should fix this problem.

Because of the way the Ultra C/C++ linker works, in the interests of keeping module sizes small, no pre-instantiated template classes have been provided in the Standard Library. This results in longer prelink times, but unnecessary code is not linked in as would happen with a pre-instantiated version. I-code linking can be used with pre-instantiated classes with lesser time spent prelinking, but the time gains in instantiation are usually dominated by the longer optimization and code generation times.

Prelinker

The prelinker is now a separate compiler phase and can be passed options using the `-pl=<option>` syntax on the `xcc` command line. One of the most useful options is `-pl=S`, which causes the prelinker to recompile all sources with generation of instantiation flags suppressed. This results in smaller ROFs, especially when the ROFs are to be archived into a library. The `-pl=q` option can be used to suppress prelinker messages.

Frequently Asked Questions about Migrating to Ultra C++ v2.1

The main hurdles in migrating from Ultra C++ 1.3 to Ultra C++ 2.1 are encountered in these areas:

1. Header files, specifically `<iostream>`
2. Memory allocation
3. Templates
4. Namespaces



For More Information

The following sections offer additional information on related topics:

- [Headers](#)
 - [New Template Features](#)
 - [Namespaces](#)
-

The problems/questions usually encountered are:

Q: The compiler complains about `cout`, `fstream`, etc. being undefined. Why?

This happens because you are using `cout` without any qualifications. Most library names now are placed in the namespace `std` which itself is placed in the global namespace. Thus, a quick fix is to place the directive:

```
using namespace ::std;
```

at some point before referring to any standard C++ names. This declaration injects all names in the named namespace into the current scope. In that way, if the `using` directive is at file scope, the names are available in any part of the program. On the other hand, if the `using` directive is inside a block, the names are only available inside the block, e.g.,

```
#include <iostream>

int main(int argc, char ** argv)
{
    if (argc > 1) {
        using namespace ::std; // limited to if-block
        cout << argv[1] << endl;
    }
    else
        cout << "no args"; // error: cout unknown
}
```

A quick solution for compiling old code is to create files with `.h` extensions and then include the newer files from these. For example, creating an `<iostream.h>` with the contents:

```
#include <iostream>
using namespace ::std;
```

allows old code to compile. However, be warned that placing `using` declarations in headers is not recommended; naming conflicts and ambiguities can occur.

Q: When I forward declare a standard C++ class such as `ostream`, I get redefinition errors. Why?

Assuming that you are accessing namespace `std`, the following is an error in modern C++:

```
#include <iostream>
class std::ostream; // causes redefinition error
```

This is because `iostream` classes (and most other classes) are template classes, e.g., `std::ostream` is really `std::basic_ostream`. To keep compatibility with old code, `std::ostream` is made available as a `typedef`. If you have any need to forward declare classes in the `iostream` hierarchy, use the header `<iosfwd>`.

Q: I tried using the `streambuf` member function `seekoff()` and got the error "unaccessible member function". Why?

ANSI has changed several member functions of traditional `iostreams`, either moving them to protected access or renaming them. In the above case, use `pubseekoff()`. Similarly, `seekpos()` has been made protected and a public version is available as `pubseekpos()`.

Q: What are the `.ii` files and do I need to delete them before each `os9make`?

The `.ii` files are for recording instantiation information for templated entities in your programs. Every C++ file containing references to templated entities generates a `.ii` file. Only in rare circumstances should these be deleted; even if you change your sources completely, do not delete these files. Deleting them will potentially lengthen pre-link times; there are no other positive effects. Two of the rare circumstances in which you should delete these files are during a build of libraries containing templates or after installation of a new version of Ultra C++. In this case, all `.ii` files for the client programs should (in general) be deleted before attempting to compile/link with the newly created or installed library.

Q: I'm trying to use `_os_intercept()` but I can't use a signal handler like

```
void sighandler(int signo);
```

The compiler complains about incompatible function pointers. What do I do?

Use the following method:

```
extern "C" typedef void (*PFV)();
extern "C++" void sighandler(int); // can also be extern "C"
//...
_os_intercept(PFV(sighandler), ...);
```

Note that there is no way to say, "any function returning `void` can be passed as the first argument to `_os_intercept()`" in C++. Static type checking forbids this.

Q: What is RTTI? And how would I use it?

RTTI, or **R**un **T**ime **T**ype **I**dentification, allows type checking at run time for cases where type checking at compile time is impossible or difficult to do. RTTI is available in the form of the new cast `dynamic_cast` and the `typeid` operator. Whereas, previously, C++ code was limited to using unsafe old style casts to cast from base classes to derived classes, newer code can do much better:

```
class Base { virtual void f(); /* ... */ };
class Derived : public Base {
    virtual void f();
    virtual void g();
    /* ... */
};

Base* bp = get_a_pointer(); /* returns pointer to derived object */
Derived *dp;
dp = dynamic_cast<Derived*>(bp);
if (dp == 0) {
    bug();
}
dp->g();
```

Note that `((Derived*)bp)->g()` could result in a catastrophic program error if `bp` is not pointing to a derived object. `dynamic_cast` works on references too, but on failure to cast, throws a `std::bad_cast` exception.

The `typeid` operator returns an object of type `std::type_info` defined in the header `<typeinfo>`. Among other member functions, `std::type_info` has the member function `name()` which returns a pointer to a static null terminated character array denoting the name of the type. It can be used for implementing persistent objects.

In general, `dynamic_cast` failure should be treated as a program logic error. This means that a properly working program should never have a `dynamic_cast` which fails. Of course, it is possible to use `dynamic_cast` or `typeid` to find the dynamic type of an object and then take a branch based on the type, but that negates the whole point of having virtual functions.

If speed is an issue, it is probably better to use `static_cast` to navigate class hierarchies. However, this can be done after using `dynamic_cast`.

Q: Why does my C++ program abort unexpectedly with error code 177 (unexpected or bad signal)?

Program aborts, or terminations via the `std::terminate()` function, can be caused by uncaught exceptions, or exception specification violations. To catch both kinds of errors, define and set handlers.

Example code follows:

```
#include <exception>
void terminate_handler()
{ /* ... */ }
void unexpected_exception_handler()
{ /* ... */ }

using std::set_unexpected;
using std::set_terminate;
int main()
{
    set_unexpected(&unexpected_exception_handler);
    set_terminate(&terminate_handler);
    /* ... */
}
```

This way, if either of the standard C++ functions `std::terminate()` or `std::unexpected()` are called by the run-time system, control is transferred to the handlers.

Another way in which a C++ program terminates by calling `std::terminate()` is when a destructor called during the processing of a thrown exception, itself throws an exception. For example,

```
class Object {
public:
    /* ... */
    ~Object() throw (error_type)
    {
        /* ... */
        if (error_condition)
            throw error_type();
    }
};

void f() throw (error_type)
{
    Object obj;
    /* ... */
    if (some_error_condition)
        throw error_type();
}
```

Due to the thrown exception in `f()`, the variable `obj` needs to be destroyed properly by calling its destructor. However, if this destructor itself throws an exception, say, to report some error, then the program terminates via `std::terminate()`. To prevent this from happening, destructors which can throw exceptions should first call the standard C++ function `std::uncaught_exception()` to check whether the destructor is being called as part of exception processing. If true, then no exceptions should be thrown from the destructor. Thus, `Object()` should be

```
#include <exception>

Object::~~Object() throw (error_type)
{
    /* ... */
    if (error_condition && !std::uncaught_exception())
        throw error_type();
}
```

Also, if exceptions have been disabled and no exception handler function has been set by using `__mw::set_exception`, an error condition in the standard library which requires throwing of an exception results in calling the default handler. This results in aborting the program. To prevent this, define an exception handler function as described in the [Compiling with Exceptions Disabled](#) section of [Chapter 12](#) in *Using Ultra C/C++*.

Q: My current code checks for return value 0 from operators `new/new[]`. How do I migrate to the newer versions which throw exceptions?

The standard operators `new` and `new[]` now throw exception `std::bad_alloc` (defined in `<new>`). To port old code which checks for a null return value, one of several techniques can be used:

1. If you are using `new/new[]` to allocate user defined class objects, then assuming your class is called `My_Class`, define the allocation operators as class members:

```
#include <new> /* add this header */

class My_Class {
public:
/* add these member functions */
    static void* operator new(size_t nbytes)
    {
        using std::bad_alloc;
        try {
            void* p = ::operator new(nbytes);
            /* above will throw on failure */
            return p;
        }
        catch (const bad_alloc& error) {
            return 0;
        }
        catch (...) {
            throw; // something else; rethrow
        }
    }
/* similarly for operator new[] */
};
```

Make sure that the signature matches that of the global allocation operators as defined in `<new>`. Then, whenever allocation is done for objects or arrays of objects of `My_Class`, the user defined allocation operators are going to be called which behave as the older versions. The rest of code can remain unchanged.

If exception handling is disabled, the `new` operator in the above class can be written as:

```
#include <new>
class My_Class {
public:
    static void* operator new(size_t nbytes)
    {
        return ::operator new(nbytes, std::nothrow);
    }
    /* ... */
};
```

The "nothrow" version of operator `new` is invoked by passing it the predefined constant `std::nothrow` in header `<new>`.

If exceptions are disabled and allocation failure occurs, the program is notified of the exception, which results in the default action of program termination (handling conditions requiring throwing of exceptions when exceptions are disabled as described in [Chapter 12](#), section [Compiling with Exceptions Disabled](#) of *Using UltraC/C++*).

A general way to prevent throwing of `bad_alloc` — or calling the user defined exception handling function when exceptions are disabled — upon allocation failure is to define and set a `new_handler` function. This is called whenever allocation failure occurs. This handler is supposed to make more memory available for allocation or else throw a `bad_alloc` or never return. With exceptions disabled, one can report memory allocation failure in this way and never have to check for `new` return values. For example:

```
#include <new>
using std::set_new_handler;

void out_of_memory_handler()
{
    // ...
}
```

```
    }  
    int main()  
    {  
        set_new_handler(&out_of_memory_handler);  
  
        //...  
    }
```

2. If your code is allocating built-in types, the you'll have to enclose your `new`-expressions in `try/catch` blocks or else change the allocation to use the "nothrow" version of `new`. `try/catch` blocks need not enclose each `new`-expression. As a first step, you can enclose the start of the function call graph in `main()` in the `try/catch` blocks. This way, if an exception is thrown, it eventually reaches that point. This is, however, unsatisfactory for locating the failure point. A better method is to enclose whole function bodies in `try/catch` blocks.

Q: My template specializations won't compile anymore. Why?

This is because the new compiler demands the new template specialization syntax. Now, each specialization must be preceded by `template<>` to tell the compiler that what follows is a specialization. However, the older syntax is still supported via the compiler option `-fe=-old_specializations`.

Q: Some of my old code such as that shown below won't compile anymore. Why?

```
template<class Object> class Container {  
public:  
    Object::Address_Type  obj_address;  
    /* ... */  
};
```

The keyword `typename` is now needed to indicate to the compiler that a name denotes a type. Thus, the above should be written as:

```
template<class Object> class Container {  
public:  
    typename Object::Address_Type  obj_address;  
    /* ... */  
};
```

Old code can be compiled without change by using the compiler option `-fe=-implicit_typename`.

Q: How do I turn off exceptions (these are turned on by default in the new compiler)?

Use the `-qnx` option. This suppresses generation of exception handling code and link with "no-exception" versions of the library. This option also enables library extensions for handling error conditions in the library (described in [Chapter 12](#) of *Using Ultra C/C++*).

Q: How can I make the compiler issue more diagnostics (apart from warnings and errors)?

Use the `-fe=-remarks` option.

Q: When will exception processing code be added to my C++ functions?

The compiler generates additional exception handling code for a function if

- A `try/catch` block is present in the function}
- A throw-expression is present in the function
- A variable of a class with a user defined destructor is declared in the function **and** exceptions are **not** turned off.

To prevent exception handling code from being generated, simply use the `-qnx` option. Note that this option does not allow one to use any exception features at all including exception specifications for functions.

Q: Is partial specialization supported?

Not at this time.

Q: What are the new style casts for? Where should they be used?

Ultra C++ 2.1 now supports the new-style casts for better type safety in C++ programs. These casts were introduced to plug the type safety hole that the old-style cast makes in the C++ type system. Four new casts have been defined to replace the "old" C-style cast (which is still supported):

- `static_cast`
- `const_cast`
- `reinterpret_cast`
- `dynamic_cast`

The first three casts are purely compile-time casts. The compiler rejects them if they violate language rules. The fourth cast, however, is a run-time cast and generates calls to the run-time library.

`dynamic_cast` was discussed above; the other three casts are explained below:

`static_cast` is a conversion cast which performs language-defined conversions. It can be used, for example, to convert between the various primitive types like `int` and `float`, or to navigate the class hierarchy. Though static casts can be used to cast a pointer to a base class to a pointer to a derived class, `dynamic_cast` is recommended for that operation, unless the pointer to the base class is guaranteed to be pointing to a derived class object. `static_cast` also works on references, and can be used to invoke a user defined conversion function. An example is:

```
class String {
public:
    /* ... */
    operator const char* ();
};

/* ... */
String s;
// now invoke conversion function
const char* cstr = static_cast<const char*>(s);
```

`const_cast` can be used to cast away "constness" or "volatileness" where required. All correctly written C++ code should almost never require this cast, but where interfacing with legacy code, especially C code, it can come in useful. For example,

```
extern "C" copy(char*, char*, int); // legacy C function

String s; // from above example
const char* cst = static_cast<const char*>(s);
char buf[MAX];
copy(buf, const_cast<char*>(cstr), sizeof buf);
```

Be warned, however, that casting away "constness" indiscriminately can lead to memory access violations as `const` data can be placed in read-only memory by the compiler. Similarly, casting away "volatileness" can lead to incorrect results.

For cases where implementation defined casting is needed, `reinterpret_cast` can be used. Examples are converting integral values to pointers and vice versa, pointers to one type to pointers to a different type, interpreting bit-patterns as data values. For example,

```
// integral value to pointer
class Object;
Object *ptr = reinterpret_cast<Object*>(0xffffffff00);

// function pointers (including C & C++)
extern "C" typedef void (*CPFV)();
extern "C++" void Cplusplus_function(int);
// force assignment though both type and linkage
// is different
CPFV cfnptr = reinterpret_cast<CPFV>(&Cplusplus_function);

// pointer types
extern copy(char*, char*, int);
int iarray[MAX];
char s[MAX * sizeof(int)];
copy(s, reinterpret_cast<char*>(iarray), sizeof s);

// interpretation of bit patterns
// old way: void* p = ...; Type value = *(Type*)p;

// define IEEE infinity value for double
unsigned inf_pat[2] = { 0x7ff00000, 0x00000000 };
double infinity =
```

```
*reinterpret_cast<double*>(&inf_pat[0]);  
  
// or using references  
double infinity =  
    reinterpret_cast<const double*>(inf_pat);
```

The advantage of the new-style casts over the old one is that they are more readily visible in code and can be searched for easily. Old-style casts are difficult to spot or search for in code. Spotting implementation-defined cast operations when porting is quite useful; for example, if a file contains `reinterpret_casts`, it needs to be examined carefully to find out what those casts are doing, and if necessary to change those casts accordingly to reflect the move to a different system. `dynamic_cast` can be used to find program (logic) bugs which might otherwise go undetected with the old-style cast. Refer the ANSI C++ Standard to find out the exact definition of these casts.

Some Observations

The ANSI C++ Standard has been approved only recently. Consequently, experience with the newer features is limited. Some features are not yet implemented in Ultra C++ but will be in future releases. However, given the nature of constraints that embedded systems place, particularly on size, it is best to use only a subset of C++. For example when exceptions are disabled and multiple inheritance is not used, C++ code can give performance equivalent or very close to that of C code. Templates, if used correctly, can be used to write efficient code. The Ultra C++ template mechanism instantiates only that which is necessary. Future versions of Ultra C++ will optimize this further. User-defined allocation operators can allow clean custom memory management. With placement `new`, which allows one to construct objects at arbitrary addresses, a feature quite useful in embedded systems, Ultra C++ has rich memory management facilities which are cleaner than, and as efficient as, those of C.

Index

Symbols

__obj_assign(x) 60
 __obj_constant(x) 62
 __obj_copy(x) 61
 __obj_modify(x) 59
 __STDC__ 115
 __cplusplus 115
 _ANSI_EXT 114
 _asm()
 Pseudo Function 50
 Statements 53
 _asm() Statements 50
 _asm() Statements, Avoiding Hard-Coded Labels 76
 _asm() Statements, Embedded 54
 _BIG_END 114
 _LIL_END 114
 _OS9000 114
 _OS9THREAD 114
 _OS9THREAD_UNSAFE 114
 _SPACE_FACTOR 115
 _TIME_FACTOR 115
 _UCC 115

Numerics

68K Processor Library Functions 99

A

Absolute Symbols
 Both -r and -c Options 369
 -r Option without -c Option 369
 additional library files 148

align 312
 Align to Even Byte Boundary 312
 allocate
 additional size for program stack 139
 additional stack space to specific phase 139
 Allocating Registers in Embedded Assembly 81
 Anachronisms Accepted 425
 ANSI
 compile in ANSI extended source mode 128
 ANSI- and ISO-Compliance 16
 ao 154
 Arguments 282
 Assembler 212, 224, 263, 513
 assembler 117
 Assembler and Object Code Linker Overview 223
 Assembler Options, Set 296
 Assembler, Running 264
 Assembly
 Language in C Source Files 48
 Language Program Development Process 229
 Level Optimizations 409
 Listing Format 278
 Optimizer 210, 510
 Valid 241, 251
 assembly
 include C source 129
 optimizer 117
 phase suppress 128
 Assembly Language Program Development Process 229
 Assign
 Offset Counter Value to Label 317
 Value to Symbolic Name 288, 302
 Assignment Translation 397
 Attribute/Revision (2 bytes) 241, 251
 Automatic Instantiation 326
 Automatic Internal Labels 284
 Avoiding Hard-Coded Labels in `_asm()` Statements 76

B

Back End 208, 503

back end [117](#)
 Begin New Page in Listing [298](#)
 Bit Field Relationship to Local Offset [261](#)
 Building C++ Libraries Referencing Template Entities [339](#)
 Byte Numbering [239](#)

C

C

- Expressions [56](#)
- Keywords and Struct Member Names [97](#)
- C Language Features [414](#)
- C++
 - Features and Restrictions [92](#)
 - Language Features [421](#)
- c89
 - behavior [121](#)
 - executable output file [122](#)
 - file name extensions [121](#)
 - library specifications [123](#)
 - option mode [106](#)
 - option parsing [121](#)
- c89 Executive Mode Compiler Phase Codes [181](#)
- CC [107](#)
 - environment variables [106](#)
- cc Executive Option Mode Compatibility and Defaults [35](#)
- change
 - executive mode [107](#)
- Changes to Rogue Wave Tools++ Libraries [438](#)
- Characters [417](#)
- CLIB [112](#)
- Code
 - Motion and Combining [398](#)
 - Size [39](#)
 - Speed [42](#)
 - Symbols
 - with Both -r and -c Options [368](#)
 - with -c Option [367](#)
 - with -r Option, without -c Option [367](#)
 - without -c Option [365](#)
- code

- area
 - const qualified pointers in 129
- Code Comments, Treatment 14
- com 313
- command line options 130, 151
 - allocate
 - additional size for program stack 139
 - additional stack space to specific phase 139
 - ANSI
 - compile in 128
 - compile
 - in version 3.2 compatibility mode 128
 - strict ANSI mode 128
 - define name and value for preprocessor 131
 - force compiler to suppress vsect directives 141
 - generate trap instructions to access floating point math routines 153
 - get option information about specific phases 127
 - include
 - C source code as comments in generated assembly files 129, 130
 - I-code versions of standard library on I-code link line 136
 - library in object code link phase 137
 - link program with C shared library 135
 - list target processor-specific options 148
 - optimize 142
 - output two symbol modules for debugging 134
 - override output file naming conventions 133, 134, 143
 - pass
 - arguments to specific phases of compilation process 152
 - option to specified phase 127
 - options to linker 138
 - place temporary files in specified directory 149
 - preprocess to file 144, 145
 - preprocess to standard out 131
 - prevent linker from creating jump table 136
 - prevent use of default libraries during linking 141
 - print
 - arguments passed to phases 129
 - help information about optimization levels 142
 - source code as comments with assembler code 129

- set
 - edition number 131
 - optimization level 143
 - sticky bit in module header 128
- specify
 - additional directory to search for preprocessor #include files 135, 151
 - alternate root psect 130, 145
 - directory
 - default library files 153
 - for intermediate step files 150
 - user library files 137
 - library file for search 154
 - output module's name 141
 - phases to skip 154
 - quiet mode 146
 - target operating system by name 150
 - target processor and sub-options 150
- stop generating stack-checking code 147, 148
- suppress
 - assembly phase 128, 147
 - inclusion of default libraries in object code link phase 136
 - linking modules into executable programs
 - library 147
 - ROF 129
 - phase execution 135
 - symbolic debugging information 147
- undefine
 - previously defined preprocessor macro names 151
- verbose command line output 128
- weight
 - given to space considerations 141, 148
 - given to time considerations 139, 149
- Comment Field 277
- Common Predecessor Code 399
- Common Subexpressions 389
- Common Successor Code 398
- Common Tail Code 399
- compatibility
 - option mode 106
- Compatibility with the Microware K&R C Compiler 94

- compile
 - strict ANSI mode 128
- Compiler
 - Components 19
 - Features 16
- Compiler Components 20
- Compiler Makefiles, Running 95
- Compiler Phase
 - Options 179
- compiler phase
 - get option information for 127
 - pass arguments to 152
 - pass options to 127
 - specify endpoint of compilation 132
- Compiling
 - 37
 - Exceptions Disabled 434
 - Single C Source File 25
 - Single C++ Source File 26
- Compiling Multiple
 - C Source File Applications 29
 - C++ Source File Applications 30
- Conditional Assembly 291
- const 90
- Constant
 - Folding 385
 - Operand 257
 - Propagation 384
 - Sharing 405
- Constant Data
 - 0 bytes 256
 - Section 256
- Convert Microware K&R Style Assembly Language to Ultra C/C++
 - 374
- Create Libraries
 - and Display Library Information 380
- Creating
 - C++ Libraries Containing Templates 334
 - Libraries
 - Reference Templated Entities 335

D

data
 area
 perform layout 141
Data Symbols
 with -c Option 368
 without -c Option 366
Date/Time Assembled (6 bytes) 241, 251
dc 315
deasm 374
Debug Information
 variable length 245, 256
Debug Information Section 245, 256
Declarators 419
decode 376
Decrement Offset Counter, Then Assign Value to Label 321
Defaults 39
Define Constant 315
define name and value for preprocessor 131
Define Storage 319
Description and Example 330
Dialect Accepted 421
Differences Between Compatibility Source Mode and the Microware
 K&R C Compiler 96
Directive Statements 285
Directive Statements 285
do 317
ds 319
Dump Symbol Information from I-Code Files 377
dz 320

E

Edition number
 2 bytes 241, 251
edition number 131
Embedded _asm() Statements 54
emplate 334
End
 Program 287

- end 287
- Environment
 - 415
- environment variables
 - CC 107
 - CLIB 112
 - MWOS 113
- equ 288
- Error
 - Checking 97
 - Messages 81
- Evaluating Expressions 267
- Examine Contents of Library Files 382
- Example Compilations 157
- Example Program Layout for a Motorola OS-9 for 68K Program
 - 235
- Exceptions Thrown
 - from C++ Operations 437
 - from the Standard C++ Library 435
- Executable
 - Memory Module 353
- Execution 350
- Executive 23, 97, 444
- executive
 - changing 107
- Executive Option Mode Compatibility 35
- Executive Option Modes 24
- Expression
 - Involving External Symbols 272
 - Operands 268
 - Operators 270
 - Operators 270
 - Tree Count 257
 - Tree Data Section 257
 - Tree Structures 257
- Extensions Accepted 426
- External _asm()
 - Pseudo Function 50
 - Statements 53
- External Definition Count
 - 2 bytes 243

- 4 bytes [254](#)
- External Definition Section [243](#), [254](#)
- External Definitions
 - variable [243](#), [254](#)
- External Reference
 - Definition Section [246](#)
 - Section [364](#)
- External References
 - variable [246](#)
- External References Count [246](#)
- Externally Reference Count [256](#)
- Externally Referenced
 - Symbol Data Section [256](#)
 - Symbols [256](#)

F

- fail [290](#)
- First Phase [351](#)
- Floating Point [418](#)
- Floating Point Math [47](#)
- force compiler to suppress vsect directive [141](#)
- Frequently Asked Questions about Migrating to Ultra C++ v2.1 [554](#)
- Front End [183](#), [450](#)
- front end [117](#)
- Function Inlining [406](#)
- Function Parameters [38](#)

G

- get
 - option information about specific phases [127](#)
- Global
 - Label Overrides [370](#)
 - Stack Checking Variables [84](#)
- Global Definition
 - Entries [359](#)
 - Hash Table [359](#)
 - Section [359](#)

H

Hardware Emulation [47](#)
Header Expansion (4 bytes) [253](#)
Header Section [240](#), [250](#)
Headers [544](#)

I

I-code
 optimizer [117](#)
i-code
 linker [117](#)
I-Code Link Method
 Compiling Multiple C Source File Applications [29](#)
I-Code Link Method
 Compiling Multiple C++ Source File Applications [30](#)
I-Code Linker [27](#), [200](#), [491](#)
I-Code Linking [40](#)
 Makefile
 Example Directory Structure [165](#), [166](#)
 Partial Applications Example Directory Structure [170](#)
I-Code Optimizer [202](#), [498](#)
Identifiers [416](#)
idump [377](#)
if...else...endc [291](#)
Ifxx Statement Descriptions [292](#)
il [154](#)
Implicit Inclusion [433](#)
include
 C source code as comments in generated assembly files [129](#),
 [130](#)
 I-code versions of standard library on I-code link line [136](#)
 library in object code link phase [137](#)
Initial Loop Condition Testing [401](#)
Initialized Data
 variable length [245](#), [255](#)
Initialized Data Section [245](#), [255](#)
Initialized Remote Data
 variable length [245](#), [255](#)
Inlining External Assembly Functions [72](#)

Input File Format 273
 Insert Blank Lines 304
 Instantiation
 #pragma Directives 431
 Mode Command Line Options 430
 Modes 430
 Pragma Arguments 431
 instantiation 334
 Integers 418
 Integration with Existing Microware Products 17
 Internal Reference Section 363
 Internal Reference Section 363
 Invariant Hoisting 402
 Invoke trap handler 324
 io 154

J

jump table
 prevent linker creation 136

K

Keywords 89

L

Label
 Fields 273
 Pseudo Functions 68
 Language 413
 Language Features 413, 537
 C++ Standard but Not Accepted 542
 Libgen 355
 libgen 380
 Library
 File 355
 Functions for All Other Processors 100
 Header 357
 Header 357

- library
 - include
 - I-code versions of standard library on I-code link line 136
 - library in object code link phase 137
 - link program with C shared library 135
 - prevent use during linking 141
 - prevent use of default libraries during linking 141
 - specify
 - directory
 - default library files 153
 - library file for search 154
 - location of standard MWOS 141
- library files
 - specify additional 148
- Library Format
 - Created by libgen 357
 - Type 1 psect Information Entries 360
 - Type 3 psect Information Entries 362
- Lifetime 387
- link
 - program with C shared library 135
- Linker 226
 - Defined Symbols 365
 - Names 228
- linker
 - do not create jump table 136
- Linking Code for Non-OS-9 Systems 372
- list
 - target processor-specific options 148
- Listing Title, Rename 305
- lo 321
- Local Reference
 - Section 247
- Local References
 - Count 247
 - variable 248
- Location
 - Counters 237
 - Tracking 411
- Loop
 - Optimizations 401

Rotation [386](#)
 Unrolling [403](#)

M

Macro [279](#)
 Assembler Names [225](#)
 Definition [294](#)
 macro
 endm [294](#)
 main() Function Parameters [38](#)
 Maximizing Speed [45](#)
 Memory Module [226](#)
 Merged Libraries [355](#)
 Merging Common Tails [409](#)
 Messages [443](#)
 Methods for Reducing Compiled Code Size [39](#)
 Microware
 Version 3.2 K&R C Compiler [14](#)
 Migrating to Ultra C++ version 2.1 [535](#)
 Minimizing Space [46](#)
 -mode [106](#), [107](#)
 Mode Descriptions [190](#)
 Module
 Header Override Symbols [370](#)
 Motorola OS-9 for 68K Program [235](#)
 mplied [334](#)
 Multiple
 Copies of Some Library Functions [98](#)
 ROF Memory Module [227](#)
 Source File
 I-Code Library
 Creation
 with Makefile [175](#)
 Multiple Source File
 I-code Library
 Creation
 Example Directory Structure [177](#)
 Non-Program I-Code Linked Example Directory Structure [172](#)
 Non-Program I-Code Linked with Makefile [171](#)

Object Library Creation Example Directory Structure 174
 Object Library Creation with Makefile 173
 Program I-Code Linked 160
 into Segments with Makefile 167
 with Makefile 162
 Program Object Code Linked 161
 with Makefile 166
 MWOS 113
 location 141

N

nam 295
 Name
 Clashing 97
 Demangler 376
 Name of Module 243, 253
 Namespaces 545
 New
 Language Features 537
 Template Features 549
 Non-Program Modules 84

O

obj_assign(x) 60
 obj_constant(x) 62
 obj_copy(x) 61
 obj_modify(x) 59
 Object
 Size and Alignment 93
 Usage Pseudo Functions 58
 object code
 link 117
 Object Code for the Module
 variable length 244, 255
 Object Code Link Method
 Compiling Multiple C Source File Applications 32
 Compiling Multiple C++ Source File Applications 33
 Object Code Linker 31, 217, 343, 524

Object Code Section 244, 255
 Offset
 Entry Point 242, 252
 Offset Counter Origin, Set 323
 Offset to Uninitialized Trap Entry Point 242, 252
 Offsetting Stack Changes 411
 Operand Field 277
 Operation Field 276
 Operators new and delete 547
 opt 296
 Optimization 383
 Options 42
 optimizations
 print
 help information 142
 set level of 143
 optimize 142
 Optimizing 17
 options
 read from file 154
 org 323
 OS-9/80x86 Processor Library Functions 99
 output
 two symbol modules for debugging 134
 override output file naming conventions 133, 134, 143
 Overview 13

P

pag 298
 parameter
 read from file 154
 pass
 arguments to specific phases of compilation process 152
 option to specified phase 127
 specified options to linker 138
 Passing Options 180
 Pipeline Scheduling 411
 place
 temporary files in specified directory 149
 Pointer Tracking with CSE 390

Prelinker [215](#), [325](#), [329](#), [522](#), [554](#)
 Execution [330](#)
 prelinker [117](#)
 Preprocessing Directives [420](#)
 preprocessor
 predefined macro names [114](#)
 undefine previously defined macro names [151](#)
 print
 arguments passed to phases [129](#)
 help information about optimization levels [142](#)
 source code as comments with assembler code [129](#)
 Process Data Area [354](#)
 Processor-Specific Information, Treatment [14](#)
 Program Layout for Motorola OS-9 for 68K [235](#)
 Program Section [299](#)
 Program Section Declarations
 psect and vsect [231](#)
 psect [299](#)
 Section [360](#)
 psects [231](#), [345](#)
 Pseudo Functions, Designator [63](#)
 Pseudo-Instructions [310](#)
 Pseudo-Instructions [310](#)
 Purpose [344](#)

Q

Qualifiers [419](#)

R

rdump [382](#)
 Reducing Branch Complexity [410](#)
 Reference
 6 bytes [257](#)
 Count 4 bytes [261](#)
 Data Section [261](#)
 Structures [261](#)
 Referencing Objects Directly [77](#)
 Referencing Template Entities [339](#)

Register
 Coloring and Coalescing 388
 Designator Pseudo Functions 63
 Relocatable
 Object File Format 238
 Program Sections 230
 remote 91
 Remote Constant Data
 0 bytes 256
 Remote Constant Data Section 256
 Remote Initialized Data Section 245, 255
 Rename Listing Title 305
 Rename Program 295
 Repeat Assembly Sequence 301
 rept ... endr 301
 Reserve
 Memory for Common Block 313
 Zero Bytes 320
 Return Error Message 290
 ROF Edition Number 15
 Format 250
 Operator Codes 259
 ROF Edition Number 9
 Format 240
 ROF Sync Bytes
 4 bytes 240, 250
 Running Compiler Makefiles 95
 Running the Assembler 264

S

Sample _asm() Statements 50
 search for
 include files 111
 library files 112
 Second Phase 352
 Set
 Assembler Options 296
 Offset Counter Origin 323
 set 302
 edition number 131

- optimization level 143
- sticky bit in module header 128
- Single Source File Program 158
 - I-Code Linked with I-Code Libraries 159
- Size
 - Constant Data 252
 - Debug Information 243, 253
 - Initialized Data 241, 251
 - Object Code 242, 252
 - Parameter, Using 71
 - Remote Constant Data 253
 - Remote Initialized Data 242, 253
 - Remote Static Storage 242, 252
 - Stack Required 242, 252
 - Static Storage 241, 251
- Small Library Function 40
 - Differences 41
- Software Emulation 47
- Some Observations 566
- Source Code Compatibility 34
- Space and Time Control 44
- Span Dependent Optimizations 408
- spc 304
- specify
 - additional directory to search for preprocessor #include files
 - 135, 151
 - alternate root psect 130, 145
 - directory
 - default library files 153
 - for intermediate step files 150
 - user library files 137
 - library file 154
 - output
 - module's name 141
 - phases to skip 154
 - quiet mode 146
 - target operating system by name 150
 - target processor and sub-options 150
- Stack Checking 84
 - Functions 85
- Standard C++ Library 551

Standard I/O and Microware K&R C Compiler ROFs 102
 Statements 419
 stop generating stack-checking code 147, 148
 Strength Reduction 403
 String Table 360
 Structure 280
 Structures, Unions, Enumerations, and Bit-Fields 419
 Structuring and Building Template Libraries 340
 Suggestions Related to Using `_asm()` Statements 72
 suppress
 assembly phase 128, 147
 inclusion of default libraries in object code link phase 136
 linking modules into executable programs
 library 147
 ROF 129
 phase execution 135
 symbolic debugging information 147
 Symbol References External to the Library 364
 Symbolic Names 266
 Symbols Types 348

T

Target Processor Type (2 bytes) 253
 tcall 324
 Template Features 549
 Template Instantiation 428
 Terms and Definitions 334
 Text Output 348
 Time and Space Control 44
 Translation 414
 Treatment
 Code Comments 14
 Processor-Specific Information 14
 ttl 305
 Type/Language 240, 250

U

ucc

Executive Mode Compiler Phase Codes 180
 ucc and c89 Option Mode Time and Space Options 44
 Ultra C/C++ Compatibility 94
 undefine
 previously defined preprocessor macro names 151
 Usage 346
 use 306
 Use External File 306
 Useless 395
 Code Elimination 391
 Copy Elimination 393
 Pointer Elimination 395
 Using
 Assembly Language 101
 csl 98
 deasm 101
 Libraries 100
 Remote as a Storage Class 96
 Size Parameter 71
 Time/Space Ratios 46
 Utilities 373

V

Variable
 Lifetimes 387
 Section 308
 verbose command line output 128
 volatile 89
 vssect 308
 vssects 232

W

warnings
 enable warnings 130, 151
 Weight 387
 weight
 given to considerations
 space 141, 148

time 139, 149
weighted 387
Writing "Exception-Independent" Code 438

xplicit 334

X

