IBM C/C++ Tools

S61G-1181-00

Programming Guide

Version 2.0

IBM C/C++ Tools

S61G-1181-00

Programming Guide

Version 2.0

Second Edition (March 1993)

This edition applies to Version 2.0 of IBM C/C++ Tools (Programs 61G1176 and 61G1426) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change or addition.

Requests for publications and for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative. Publications are not stocked at the address given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
21/986/844/TOR
844 Don Mills Road
North York, Ontario, Canada. M3C 1V7

You can also send your comments by facsimile to (416) 448-6057 addressed to the attention of the RCF Coordinator. If you have access to Internet, you can send your comments electronically to **torrcf@vnet.ibm.com**; IBMLink, to **toribm(torrcf)**; IBM/PROFS, to **torolab4(torrcf)**; IBMMAIL, to **ibmmail(caibmwt9)**

If you choose to respond through Internet, please include either your entire Internet network address, or a postal address.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page xv.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Programming Interface Information

This book is intended to help you create programs using the C/C++ Tools product. It primarily documents General-Use Programming Interface and Associated Guidance Information provided by the C/C++ Tools product.

General-Use programming interfaces allow the customer to write programs that obtain the services of the C/C++ Tools compiler, debugger, browser, execution trace analyzer, and class libraries.

However, this book also documents Diagnosis, Modification, and Tuning Information. Diagnosis, Modification, and Tuning Information is provided to help you debug your programs.

Warning: Do not use this Diagnosis, Modification, and Tuning Information as a programming interface because it is subject to change.

Diagnosis, Modification, and Tuning Information is identified where it occurs by an introductory statement to a chapter or section.

Trademarks and Service Marks

The following terms used in this publication are trademarks or service marks of IBM Corporation in the United States or other countries. They are denoted by an asterisk (*) when they first appear in the text.

BookManager	OS/2
C/2	Personal System/2
C Set/2	Presentation Manager
Common User Access	PS/2
CUA	SAA
IBM	Systems Application Architecture
Operating System/2	WorkFrame/2

The following terms used in this publication are trademarks or service marks of other corporations in the United States or other countries. They are denoted by a double asterisk (**) when they first appear in the text.

Microsoft	Microsoft Corporation
Pentium	Intel Corporation

Summary of Changes

This section summarizes the differences between the IBM* C/C++ Tools product and its predecessor, the IBM C Set/2* Version 1.0. It also describes changes made to this document S61G-1181, from the previous version S10G-4444. All technical changes to this document are marked in the text by a vertical bar in the left margin.

Changes to the Product

All C/C++ Tools components now support the C++ language in addition to C.

The following components have been added to the product:

- Execution trace analyzer (EXTRA)
- Source code browser
- Class libraries (I/O Stream, Task, Complex Mathematics, Collection Class, and User Interface)

The migration language level has been eliminated. All functions that were part of the migration libraries in the C Set/2 V1.0 product are now standard extensions.

The diagnostic messages have been revised, along with the compiler options and #pragma directives that control them, to give the user more control over which diagnostics are to be performed. Some messages have been reduced in severity from warning to informational.

Support for storing temporary files in memory (memory files) is now optional and can be enabled using the /Sv compiler option.

Support for building virtual device drivers has been added, including support for the _Far32 _Pascal calling convention and 48-bit function pointers. See Chapter 15, "Developing Virtual Device Drivers" on page 281.

Inlining of user code is supported with the /Oi option (see page 116) and _Inline and inline keywords (see the *IBM C/C++ Tools: Online Language Reference*, referred to hereafter as the *Online Language Reference*).

A keyword and a #pragma directive have been implemented to declare a function is to be exported. See the *Online Language Reference* for a description of _Export and #pragma export.

Anonymous unions are supported. See the *Online Language Reference* for a description.

An intermediate code linker has been added to the compiler. It combines all intermediate (or temporary) files for all the files specified in the icc command. The compiler then uses the combined file for optimization. See "Using the Intermediate Code Linker" on page 52 for more information.

The following compiler options have been modified:

Option	**Description**	**Page**
/K*n*	These options, which control diagnostic messages, are now mapped to the /W*grp* options. They are not supported for use with C++ files, and will not be supported at all in future releases of the C/C++ Tools product.	92
/La	Includes a layout of struct and union variables referenced by the user. In Version 1.0, it included layouts of all struct and union variables.	88
/Lx	Generates a cross-reference table of all variables referenced by the user. In Version 1.0, it generated a cross-reference of all global and external variables, plus all local variables referenced by the user.	89
/Sm	Controls compiler interpretation of 16-bit keywords. In Version 1.0, it set the language level to migration to allow all migration constructs.	101
/Ss	For C files only, allows use of double slashes (//) for comments. This option is ignored for C++ files.	102

The following compiler options have been added:

Option	Description	Page
/Fb	Produces a browser listing file.	82
/Fi	Controls creation of precompiled header files.	83
/Ft	Controls generation of header files for template functions and class declarations.	83
/Fw	Controls generation and use of intermediate files.	84
/Gh	Generates code enabled for EXTRA and other profiling tools.	112
/Gi	Generates code for fast integer execution.	113
/Gu	Passes information to the intermediate linker.	114
/Gv	Controls handling of DS and ES registers.	114
/Gw	Controls generation of FWAIT instruction after each floating-point load instruction.	115
/Gx	Removes C++ exception handling information.	115
/G5	Optimizes code for use with a Pentium** microprocessor.	115
/Lb	Includes a layout of all struct and union variables.	88
/Ly	Generates a cross-reference table of all global and external variables, plus all local variables referenced by the user.	89
/Nd	Specifies names of default data and constant segments.	116
/Nt	Specifies name of default text segments.	116
/Oi	Controls inlining of user code.	116
/Ol	Controls use of the intermediate code linker.	117
/Om	Controls size of the working set of the compiler.	117
/Op	Controls optimizations involving the stack pointer.	117
/Os	Controls use of the instruction scheduler.	117
/Pe	Suppresses #line directives in preprocessor output.	109
/Sc	Sets the language level to be compatible with earlier versions of the C++ language.	100
/Si	Controls use of precompiled header files.	101
/Su	Controls size of enum variables.	103
/Sv	Enables memory file support.	103
/Tc	Tells icc to compile the following file as a C file.	103
/Td	Tells icc to compile all following source files as C or C++ files.	104
/Tp	Tells icc to compile the following file as a C++ file.	104

Option	Description	Page
/Ts	Generates code to allow the debugger to maintain the call stack across all calls.	94
/Tx	Controls information generated when an exception occurs.	94
/W*grp*	Controls diagnostic messages.	96

The following #pragma directives have been added or modified:

checkout
: Is now mapped to #pragma info. This directive is currently valid for C files only and will not be supported in future releases.

define
: Forces the definition of a template class without defining an object of the class. Valid for C++ files only.

disjoint
: Lists identifiers that are not aliased to each other within the scope of their use. Valid for C++ files only.

entry
: Specifies entry point to the program being built.

export
: Declares a DLL function to be exported and specifies the name for the function outside of the DLL.

implementation
: Tells the compiler where to find the function template definitions corresponding to the declarations in the file including the #pragma directive. Valid for C++ files only.

import
: Specifies a DLL function to be imported using either a name or an ordinal number.

info
: Controls diagnostic messages. This directive replaces the #pragma checkout directive.

isolated_call
: Lists functions that do not alter data objects visible at the time of the function call. Valid for C++ files only.

langlvl
: No longer has the option mig to set the language level to migration. It has a new option compat for compatibility with earlier versions of the C++ language.

undeclared
: In a template-include file, separates functions that were instantiated with a declaration and those instantiated with a call. Used only by the compiler and only for C++ files.

All #pragma directives are described in the *Online Language Reference*.

The <**builtin.h**> header file has been added.

The following functions have been added to the C runtime library:

– Trigonometric and transcendental functions that exploit the 80387 processor:

_facos	Calculates arccosine.
_fasin	Calculates arcsine.
_fcos	Calculates cosine.
_fcossin	Calculates cosine and sine.
_fpatan	Calculates arctangent.
_fptan	Calculates tangent.
_fsin	Calculates sine.
_fsincos	Calculates sine and cosine.
_fsqrt	Calculates square root.
_fyl2x	Calculates *y* to the base-2 logarithm of *x*.
_fyl2xp1	Calculates *y* to the base-2 logarithm of *x* plus 1.
_f2xm1	Calculates 2 to the power of *x*, minus 1.

– Low-level functions for port input and output:

_inp	Reads a byte from an input port.
_inpd	Reads a doubleword from an input port.
_inpw	Reads an unsigned short value from an input port.
_outp	Writes a byte to an output port.
_outpd	Writes a doubleword to an output port.
_outpw	Writes a unsigned short value to an output port.

– Functions that affect interrupt procedures:

_disable	Disables external interrupts.
_enable	Enables external interrupts.
_getTIBvalue	Returns a value from the Thread Information Block (TIB).
_interrupt	Calls interrupt procedure.

– Functions that affect process control:

_threadstore Accesses a pointer to the user's thread-specific storage.

All functions are documented in the *C Library Reference*.

Changes to the C/C++ Tools Library

The C/C++ Tools library has been expanded and reorganized:

Reference information has been provided primarily online in IPF format. Guidance information is provided in hardcopy format. Hardcopy and BookManager* READ versions of the documentation can be ordered from IBM using the order form enclosed in your C/C++ Tools package.

For the most part, the online references correspond directly to hardcopy documents. The exception is the *IBM C/C++ Tools: Online Language Reference*, which combines information from both the *IBM C/C++ Tools: C Language Reference* and *IBM C/C++ Tools: C++ Language Reference* hardcopy documents.

Publications have been added for the new components as follows:

- *IBM C/C++ Tools: Execution Trace Analyzer Introduction*
- *IBM C/C++ Tools: Browser Introduction*
- *IBM C/C++ Tools: C++ Language Reference*
- *IBM C/C++ Tools: Standard Class Library Reference*
- *IBM C/C++ Tools: User Interface Class Library Reference*
- *IBM C/C++ Tools: Collection Class Library Reference*

The *IBM C Set/2 Version 1.0 User's Guide* has been renamed to the *IBM C/C++ Tools: Programming Guide*.

The *IBM C Set/2 Version 1.0 Debugger Tutorial* has been rewritten and changed to the *IBM C/C++ Tools: Debugger Introduction*. The tutorial is now online and is accessible through the debugger Help menu.

An online tutorial is also provided for the browser.

The document *IBM C Set/2 and WorkFrame/2: An Integrated Development Environment* is no longer part of the C/C++ Tools library. In its place, an online tutorial is provided with the IBM WorkFrame/2* product.

The definitions of the C language and runtime library have been placed in separate documents (the *IBM C/C++ Tools: Online Language Reference* or *IBM C/C++ Tools: C Language Reference* and *IBM C/C++ Tools: C Library Reference* respectively). All information in the *IBM C Set/2 User's Guide* (now the *IBM C/C++ Tools: Programming Guide*) and *IBM C Set/2 Migration Guide* that described language or library extensions has been moved to the appropriate reference guide.

The *SAA CPI C Reference - Level 2* (SC09-1308-02) is no longer part of the C/C++ Tools library, but you can order it separately.

Changes to this Publication

The changes made in this publication are:

- The name has been changed from the *IBM C Set/2 User's Guide* to the *IBM C/C++ Tools: Programming Guide*.
- Chapter 18, "Signal and OS/2 Exception Handling" on page 317 has been completely rewritten.
- The chapter on library functions has been moved to the *IBM C/C++ Tools: C Library Reference*.
- The chapter on language extensions to SAA (including #pragma directives) has been moved to the *IBM C/C++ Tools: Online Language Reference* and the appropriate language reference.
- Chapters have been added on the following topics:
  - Finishing your application, including binding messages and resources (Chapter 6, "Finishing Your Program" on page 123).
  - Optimizing your program for performance and for size (Chapter 10, "Optimizing Your Program" on page 165).
  - Using templates in C++ programs (Chapter 13, "Using Templates in C++ Programs" on page 225).
  - Creating virtual device drivers (Chapter 15, "Developing Virtual Device Drivers" on page 281).
  - Troubleshooting and support (Appendix D, "Solving Common C Problems" on page 401).
- The appendixes that documented error messages have been removed. Error messages are documented in the *IBM C/C++ Tools: Online Language Reference*.

Information has been added to describe the new features listed under Changes to the Product.

Minor technical and editorial corrections have been made.

Part 1. General Information

This part of the Programming Guide provides general information about the IBM C/C++ Tools product, including its features and components, installation, related publications. It also gives an example of how to compile, link, and run a short program.

General Information

Chapter 1. About This Book

This book tells you how to use the IBM C/C++ Tools product (referred to throughout the book as C/C++ Tools) to compile, link, and run C and C++ programs on the 32-bit Operating System/2* (OS/2*) operating system (OS/2 2.0 or later release).

Use this book with the other publications described in "Related Publications" on page 7.

Who Should Read This Book

This book is written for application and systems programmers who want to use the C/C++ Tools product to develop and run C or C++ applications. It assumes you have a working knowledge of the C or C++ programming language, the OS/2 operating system, and related products as described in "Related Products" on page 21.

Portability Considerations

If you will be using the C/C++ Tools product to develop C applications to be compiled and run on other Systems Application Architecture* (SAA*) systems, you should follow the SAA standards as outlined in the *SAA Common Programming Interface C Language Reference*, SC09-1308-02. If you will be using the C/C++ Tools product to develop code according to the American National Standards Institute (ANSI) standard, you should also refer to the ANSI guidelines. If you will be developing code according to the International Standards Organization (ISO) standard, refer to the ISO guidelines. General information about writing portable C code is included in the *Portability Guide for IBM C*, SC09-1405.

When following ANSI, ISO, or SAA standards, do **not** use the extensions specific to the C/C++ Tools compiler as described in the *C Library Reference*.

How to Read Syntax Diagrams

| At this time, there is no universal standard for the C++ language comparable to the C standards. If portability of your C++ programs is important, isolate those parts of your code that use the Collection and User Interface class libraries, which are specific to the C/C++ Tools product, so you can easily remove or replace them when migrating your programs.

If you will be using the C/C++ Tools product for the development of applications that will run only under the OS/2 operating system, you may want to exploit the OS/2 services and APIs and the C/C++ Tools multithread features (see Chapter 11, "Creating Multithread Programs" on page 179).

How to Read the Syntax Diagrams

This book uses two methods to show syntax. One is for commands, preprocessor directives, and statements; the other is for compiler options.

Syntax for Commands, Preprocessor Directives, and Statements

Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

The ——— symbol indicates the beginning of a command, directive, or statement.

The ——— symbol indicates that the command, directive, or statement syntax is continued on the next line.

The ——— symbol indicates that a command, directive, or statement is continued from the previous line.

The ——— symbol indicates the end of a command, directive, or statement.

Diagrams of syntactical units other than complete commands, directives, or statements start with the ——— symbol and end with the ——— symbol.

Note: In the following diagrams, STATEMENT represents a C or C++ command, directive, or statement.

How to Read Syntax Diagrams

Required items appear on the horizontal line (the main path).

```
──STATEMENT──required_item──────────────────────────────
```

Optional items appear below the main path.

```
──STATEMENT── ──┬───────────────┬── ──────────────
                └──optional_item┘
```

If you can choose from two or more items, they appear vertically, in a stack.

If you *must* choose one of the items, one item of the stack appears on the main path.

```
──STATEMENT── ──┬──required_choice1──┬── ──────────────
                └──required_choice2──┘
```

If choosing one of the items is optional, the entire stack appears below the main path.

```
──STATEMENT── ──┬────────────────────┬── ──────────────
                ├──optional_choice1──┤
                └──optional_choice2──┘
```

The item that is the default appears above the main path.

```
                ┌──default_item────┐
──STATEMENT── ──┴──alternate_item──┴── ──────────────
```

An arrow returning to the left above the main line indicates an item that can be repeated.

```
                ┌──────────────────┐
──STATEMENT── ──┴──repeatable_item─┴── ──────────────
```

A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

Keywords appear in nonitalic letters and should be entered exactly as shown (for example, pragma).

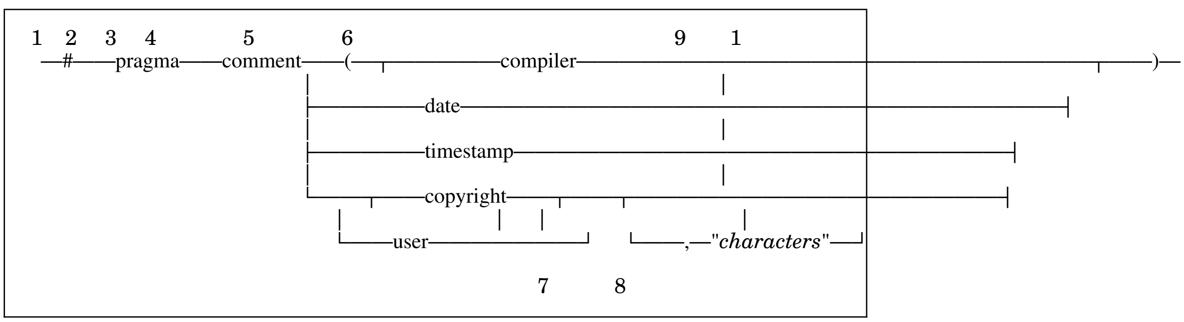
Variables appear in italicized lowercase letters (for example, *identifier*). They represent user-supplied names or values.

How to Read Syntax Diagrams

If punctuation marks, parentheses, arithmetic operators, or other such symbols are shown, you must enter them as part of the syntax.

Note: The white space is not always required between tokens, but it is recommended that you include at least one blank between tokens unless specified otherwise.

The following syntax diagram example shows the syntax for the #pragma comment directive. (See the *Online Language Reference* or *C Language Reference* for information on the #pragma directive.)



The syntax diagram is interpreted in the following manner:

1 This is the start of the syntax diagram.

2 The symbol # must appear first.

3 The keyword pragma must appear following the # symbol.

4 The keyword comment must appear following the keyword pragma.

5 An opening parenthesis must be present.

6 The comment type must be entered only as one of the types indicated: compiler, date, timestamp, copyright, or user.

7 If the comment type is copyright or user, and an optional character string is following, a comma must be present after the comment type.

8 A character string must follow the comma.

9 A closing parenthesis is required.

1 This is the end of the syntax diagram.

The following examples of the #pragma comment directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Syntax for Compiler Options

Optional elements are enclosed in square brackets [ ].

When you have a list of items from which you can choose one, the logical OR symbol (|) separates the items.

Variables appear in italicized lowercase letters (for example, *num*).

Examples

Syntax	Possible Choices
/L[+|-]	/L /L+ /L-
/Lt"*string*"	/Lt"Listing File for Program Test"

Note that, for options that use a plus (+) or minus (-) sign, if you do not specify a sign, the plus is assumed. For example, the /L and /L+ options are equivalent.

Related Publications

The following publications provide more information about the C/C++ Tools product and how to use it:

IBM C/C++ Tools: Browser Introduction, S61G-1397, shows you how to use the C/C++ Tools browser.

IBM C/C++ Tools: Execution Trace Analyzer Introduction, S61G-1398, introduces the execution trace analyzer EXTRA.

IBM C/C++ Tools: Debugger Introduction, S61G-1184, provides an introduction to the C/C++ Tools debugger.

Related Publications

IBM C/C++ Tools: Reference Summary, S61G-1441, summarizes the C/C++ Tools language syntax, reserved keywords, library functions, and compiler options.

IBM C/C++ Tools: Class Libraries Reference Summary, S61G-1186, summarizes the functions provided by the C/C++ Tools class libraries.

IBM C/C++ Tools: Installation, S61G-1363, describes the installation procedure.

IBM C/C++ Tools: License Information, S71G-1453, summarizes the features and gives warranty information.

The following reference documents provide more information about the C/C++ Tools implementation of the C and C++ languages and libraries, including class libraries. These reference documents are provided in online (.INF) format, and can be ordered in hardcopy using the order form included with the C/C++ Tools product:

IBM C/C++ Tools: Online Language Reference, DDE4LRM.INF, presents the C/C++ Tools definition of both the C and C++ programming languages. This reference includes information from both the *IBM C/C++ Tools: C Language Reference*, S61G-1399, and the *IBM C/C++ Tools: C++ Language Reference*, S61G-1185.

IBM C/C++ Tools: C Library Reference, DDE4CLIB.INF or S61G-1183, describes the C/C++ Tools C library functions.

IBM C/C++ Tools: Standard Class Library Reference, DDE4SCL.INF or S61G-1180, describes the C++ I/O Stream, Task, and Complex Mathematics class libraries.

IBM C/C++ Tools: Collection Class Library Reference, DDE4CCL.INF or S10G-1178, describes the Collection class library.

IBM C/C++ Tools: User Interface Class Library Reference, DDE4UIL.INF or S61G-1179, describes the User Interface class library.

These publications are referred to throughout this book without the IBM C/C++ Tools prefix.

Related Publications

The following publications are not included with the C/C++ Tools product, but contain information about it and may be helpful:

- *Portability Guide for IBM C*, SC09-1405, describes how to move code from one platform to another, and how to write portable code. The second edition of this document will include information on the C/C++ Tools product.
- *SAA CPI C Reference - Level 2*, SC09-1308-02, presents the SAA definition of the C language.

Additional publications that may be helpful to the C/C++ Tools user are listed in "Bibliography" on page 455.

Online Publications

The C/C++ Tools product provides online publications in two different formats, IPF and BookManager* READ.

Information Presentation Facility (IPF) Books

IPF is the online help mechanism provided by the OS/2 operating system. The C/C++ Tools product provides several online references in IPF format:

- The *Online Language Reference*, DDE4LRM.INF, is a summary of C and C++ language constructs, compiler options, and messages.
- The *C Library Reference*, DDE4CLIB.INF, describes the C library functions.
- The *Standard Class Library Reference*, DDE4SCL.INF, describes the I/O Stream, Task, and Complex Mathematics class libraries.
- The *Collection Class Library Reference*, DDE4CCL.INF, describes the Collection class library.
- The *User Interface Class Library Reference*, DDE4UIL.INF, describes the User Interface class library.

To access an online reference, use the view command. For example, to view the *C Language Reference*, at the command line in the C:\IBMC\HELP directory type:

```
view DDE4CLRM.INF
```

To get help for a specific item, type the name of the item after the file name. The system searches the table of contents and index of the *C Language Reference*. If the item exists, it opens the panel about that item. For example:

```
view DDE4HELP.INF operator precedence
```

opens the panel about operator precedence.

The Enhanced Editor (EPM) included with the OS/2 2.0 operating system has provided macros that enable it to provide context-sensitive help using the online references. To enable this help, specify the /w option when you invoke the editor:

```
epm myfile.c /w
```

To obtain help for a keyword or construct, highlight the word and press Ctrl-H. This opens the online reference at the panel for that construct.

BookManager Books

BookManager READ/2 (73F6023) is a separately orderable OS/2 product that allows you to read online documentation. It features hypertext links between related topics. You can also search documents for keywords to quickly locate the information that you need. The following publications in their entirety can be ordered in BookManager READ format, accompanied by the IBM Library Reader, which allows you to read the books without purchasing BookManager READ/2:

Programming Guide (this manual)
C Language Reference (S61G-1399)
C Library Reference (S61G-1183)
C++ Language Reference (S61G-1185)
Debugger Introduction (S61G-1184)
Execution Trace Analyzer Introduction (S61G-1398)
Browser Introduction (S61G-1397)
Standard Class Library Reference (S61G-1180)
User Interface Class Library Reference (S61G-1179)
Collection Class Library Reference (S61G-1178)

Chapter 2. Overview of the C/C++ Tools Product

This chapter summarizes the C/C++ Tools features and briefly describes the included software, compiler, runtime libraries, debugger, and the hardware and software needed. It also discusses related product offerings.

C/C++ Tools Features

Compiler and Library Features

- | Full C++ support, including templates and exception handling
- | Multithread support for the C and C++ runtime libraries
- | Support for the standard class libraries (I/O Stream, Task, Complex Mathematics)
- | Class libraries for building your own classes (Collection) and for Presentation Manager* (PM) programming (User Interface).
- Multitasking support for creating multiple processes
- Static or dynamic linking of the runtime libraries
- Fully reentrant libraries
- Ability to create user dynamic link libraries (DLLs)
- Subsystem development capabilities
- Support of low-level input/output functions
- | Exploitation of 32-bit processor features, including Pentium microprocessor support
- | Ability to inline user functions
- | Intermediate code linking for improved performance
- | Optimization including instruction scheduling, loop unrolling, and floating-point register usage
- | Generation and use of precompiled header files
- Ability to call 16-bit code from C/C++ Tools 32-bit code
- Support of callbacks from 16-bit code to 32-bit code
- Memory files for temporary storage
- Support of NaN, Infinity, and the 80-bit `long double` type as defined by Institute of Electrical and Electronics Engineers (IEEE)
- Double-byte character set (DBCS) support
- Compiler options specified on the command line or in an environment variable

- Language-level compiler options to enforce SAA or ANSI standards
- Additional features, including language and library extensions, to ease migration from other compilers.

Debugger Features

The C/C++ Tools debugger is a 32-bit source level debugger that uses the OS/2 Presentation Manager (PM) windowing services. It concurrently manages both the application and the debugger windows. The debugger provides the following features:

- Support for both C and C++ programs
- Multiple program views, including source, disassembly, and disassembly with source annotated
- Simple and complex breakpoint capabilities
- Monitor windows for local, global, and automatic variables
- Pointer and indirect referencing
- Hierarchical structure display, including nested structures
- Display of monitored variables in context
- Ability to monitor storage and the call stack
- C++ class monitors, class inheritance view, and class details notebook
- PM window analysis
- Message queue monitoring
- Tool bar for run, step, and display of registers, stack, and storage
- Display of processor and math coprocessor registers
- Support of DBCS.

Execution Trace Analyzer (EXTRA) Features

The execution trace analyzer (EXTRA) is an execution trace tool to help you tune your program's performance. EXTRA analyzes your program as it runs, and then displays the trace data in a variety of formats. EXTRA offers the following features:

- Graphical and meaningful presentation of performance information
- Support for both C and C++ programs
- Interactive display of data
- Timestamp accuracy of 838 nanoseconds per clock tick

- Ability to show sequencing of procedures across multiple threads
- Ability to cross-correlate displays
- Ability to trace calls to the operating system.

Browser Features

The C/C++ Tools browser is a program-examining tool that uses PM services to help you study your program components. With the browser you can:

- List program components by scope, kind, or attributes
- Graphically display relationships between program components, including class inheritance hierarchies and function calls
- View and edit the source code associated with a program element.

Problem Determination

In addition to the debugger and EXTRA tool, the C/C++ Tools product provides a number of problem determination aids:

- Debug memory management functions.
- Detection of possible programming errors using the /W*grp* diagnostic compiler options.
- Source code listings. These listings can include expanded macros, and the layout of structures.
- Assembler listings. These listings include annotated source.
- Precise diagnostic messages to aid problem analysis.
- Use of the intermediate code linker to detect interfile programming errors.

C and C++ Language Industry Standards

The C/C++ Tools product is designed according to the specifications of the *American National Standard for Information Systems / International Standards Organization – Programming Language C*, ANSI/ISO 9899-1990[1992], as understood and interpreted by IBM as of December 1992. Behavior that the ANSI C Standard declares as implementation-defined is described in Appendix A, “ANSI Notes on Implementation-Defined Behavior” on page 363.

The C/C++ Tools product also implements the Systems Application Architecture (SAA) C Level 2 definition, which is a superset of the ANSI standard. For more information on the SAA C standard, see the *SAA CPI C Reference - Level 2*.

At this time, there is no universal standard for the C++ language comparable to C standards. However, an ANSI committee is developing a C++ language standard. Its September 17, 1992 working paper, *Draft Proposed American National Standard for Information Systems — Programming Language C++*, X3J16/92-0060, was used as a base document for developing the C/C++ Tools C++ compiler. The C/C++ Tools C++ compiler will continue to change its design in accordance with the ANSI standard as it evolves.

Shipped Code

The following code is available as part of the C/C++ Tools product:

- 32-bit C/C++ compiler
- C and C++ runtime libraries
- Standard C++ class libraries (I/O Stream, Task, and Complex Mathematics)
- User Interface class library
- Collection class library
- Class browser
- 32-bit PM debugger
- Execution trace analyzer (EXTRA)
- Installation program
- Sample programs and tutorials
- Online help and documentation
- IBM WorkFrame/2 support
- A READ.ME file with supplemental information about the product.

Compiler

The C/C++ Tools compiler analyzes the C or C++ source program and translates the source code into machine instructions known as *object code*. The source file extension determines whether the file is compiled as a C program (.c extension) or a C++ program (.cpp or .cxx extension). (Note that by default, files with unrecognized extensions are treated as C source files.) The object code can then be linked using the LINK386 linker shipped with the C/C++ Tools product to create an *executable module* that can be run.

For more information on compiling and linking. your program, see Part 2, "Compiling and Linking Your Program" on page 27. For more information on running your program, see Part 3, "Running Your Program" on page 131.

Runtime Libraries

In addition to the standard runtime libraries, the C/C++ Tools product offers:

- Libraries for developing subsystems that do not use the runtime environment.
- Import libraries for the C/C++ Tools dynamic link libraries (DLLs).
- Libraries that allow you to create your own customized library DLLs. (See Chapter 12, "Building Dynamic Link Libraries" on page 195 for more information.)

You can link the runtime libraries to your program either statically or dynamically.

For information on the functions available in the standard runtime libraries, see the *C Library Reference*. The subsystem libraries are discussed in Chapter 17, "Developing Subsystems" on page 303 of this book.

C++ Class Libraries

The C/C++ Tools product includes 3 sets of class libraries:

Standard Class Libraries
: The Standard class libraries include the Complex Mathematics, I/O Stream, and Task Libraries. These libraries are described in the *Standard Class Library Reference*.

Collection class library
: The Collection class library is a generic C++ container class library. It provides a variety of abstract and concrete implementations of common data structures. You can use and reuse the Collection classes as "building blocks" in your programs.

User Interface class library
: The User Interface class library is an object-oriented encapsulation of PM programming. This library helps make developing PM user interfaces easier by providing a number of classes that you can use, reuse, and extend for your applications. The program model that the User Interface library is based on is more suitable to the C++ language than the traditional procedural programming model.

Browser

The browser is a PM static analysis tool that lets you look at your source code in many different ways. For example, you can display program elements such as source files, functions, and classes, and display program relationships in a graphical format.

The class browser is described in *Browser Introduction*. An online tutorial is also available to get you started with the browser, and contextual online help is provided.

Debugger

The C/C++ Tools debugger is included to help you test and analyze your code. It is a source level debugger that uses PM services. It gives you multiple views of the program, including source and disassembly code, and helps you:

- Step through a program
- Set breakpoints
- Monitor variables, storage, expressions, and stacks
- Manipulate threads.

For more information about the debugger, refer to the *Debugger Introduction*. Contextual online help and an online tutorial are also available.

Execution Trace Analyser (EXTRA)

EXTRA is a tool that analyzes your program as it runs and displays the trace data. It features a number of different graphical displays that help you identify possible performance problem areas in your code.

EXTRA is described in *Execution Trace Analyzer Introduction*. Contextual online help and an online tutorial are also provided.

Installation Program

You can install the C/C++ Tools product with the interactive installation program included with the product. The installation program allows you to choose the options you want to install and where you want to install them. It can also update your CONFIG.SYS file. Online help is provided throughout the program to assist you with the installation.

You can also use the installation program to reinstall or add new options at a later time.

For a full description of the installation procedure, see the *C/C++ Tools Installation* card.

Sample Programs

A number of examples of program code are included with the C/C++ Tools product:

- Sample source programs that demonstrate the following types of source code:

  1. A program in both C and C++ that demonstrates how to perform the same function using the different languages and how to compile, link, and run. See Chapter 3, "An Introduction to Using the C/C++ Tools Compiler" on page 23.
  2. A program to demonstrate the multithread capabilities of the C/C++ Tools compiler. See "Sample Multithread Program" on page 194.
  3. A program to build a single-thread dynamic link library (DLL). See "Creating DLL Source Files" on page 196.
  4. A program to demonstrate how to make calls to, or to be called from, 16-bit code. See "Example of Calling a 16-Bit Program" on page 294.
  5. A program to build a subsystem DLL. See "Example of a Subsystem DLL" on page 310.
  6. A program to demonstrate the use of C++ templates. See the *Online Language Reference* or *C++ Language Reference*.
  7. A program that uses the C++ exception handling facilities. See the *Online Language Reference* or *C++ Language Reference*.

- A set of make files that compile and link the above sample programs. Each sample program has two make files. One links the libraries statically; the other links them dynamically.

  Note: You must have the Toolkit installed to use the make files.

  To build a sample, use NMAKE with the appropriate make file. For example, to build SAMPLE1A, type:

  ```
  NMAKE all /f SAMPLE1A
  ```

The parameter all ensures that the entire sample is built. After you have finished with the sample, you can use NMAKE again with the parameter clean to remove all files generated by the first NMAKE command. For example:

```
NMAKE clean /f SAMPLE1A
```

removes all files generated by NMAKE for SAMPLE1A.

- A set of module definition (.DEF) files that were used to link the C/C++ Tools library DLLs and that provide an example of how to create your own .DEF files and DLLs. For more information about .DEF files, see the Toolkit documentation for the LINK386 program.
- Sample programs for the browser and debugger. See the documentation for these tools for more information about these samples.
- Sample programs for the Collection and User Interface class libraries. See the class library documentation for more information about these samples.

Online Help

The C/C++ Tools product offers two kinds of online help:

- Online references that contain information about declarations and definitions, preprocessor directives, include files, compiler options and messages, library functions, and class libraries. You can access the references through the view command. If you use the OS/2 2.0 Enhanced editor (EPM), you can get help for a particular item by putting the cursor on the item and pressing Ctrl-H.
- Contextual and overview help for the debugger, EXTRA, and browser. Help is provided to explain the various functions and tell you how to use them. You can access the help for a tool from any window within that tool by highlighting an item and pressing F1, or from the **Help** pull-down.

For more information about the online references and other online publications, see "Online Publications" on page 9.

IBM WorkFrame/2 Support

The C/C++ Tools product provides a number of files that enable it to integrate into the IBM WorkFrame/2 product Version 1.1 or higher (referred to in this book as WorkFrame/2). Among these files are the C/C++ Tools language profile, which is used to associate WorkFrame/2 projects with the C/C++ Tools product, and the compiler options DLL, which presents the C/C++ Tools options through a graphical interface. Also included are several directories of sample files for the WorkFrame/2 product, named HELLO2, GREP, MAHJONGG, PMLINES, and TOUCH.

If you install these files when you install the C/C++ Tools files (by selecting the **WorkFrame/2 support** option), the installation program creates projects and control (.PRJ) files for the samples under the WorkFrame/2 main directory.

Hardware, Software, and Operating System Requirements

The IBM C/C++ Tools product requires a workstation with a 32-bit processor (80386, 80486, or Pentium microprocessor) running the OS/2 2.0 or later operating system.

The OS/2 2.0 Developer's Toolkit, referred to in this document as the Toolkit, is also a prerequisite, primarily because it contains the system linker that the compiler uses, as well as the system header files and import libraries that increase the capabilities of the compiler, and the NMAKE utility that helps manage the build process for projects.

The IBM C/C++ Tools Version J2.0 product requires a workstation running the IBM OS/2 Version J2.0 operating system. The IBM OS/2 Developer's Toolkit Version J2.0 is also a prerequisite.

To effectively use the C/C++ Tools compiler and debugger, you need a minimum of 8M of RAM for C applications and 12M for C++ applications. You must also set your swap path to a directory with at least 10MB free for C applications or 14M for C++ applications. A full installation of the C/C++ Tools or C/C++ Tools Version J2.0 files requires about 30MB of disk space, broken down in the following manner:

Compiler and libraries	8MB
Debugger	6MB
EXTRA	2MB
Browser	2MB
Standard class libraries	.5MB
Collection class library	1MB
User Interface class library	5MB
Online information	4.5MB
WorkFrame/2 support	1MB

When you install the product, the installation program tells you how much space you have available on the selected drive and how much space is required for the options you select.

If you have an 80386 processor, an 80387 math coprocessor is recommended because it will greatly increase the speed of floating point operations. If you have an 80486SX processor, an 80487 math coprocessor is recommended.

Related Products

In addition to providing contextual help through the EPM editor and using PM services for the debugger interface, the C/C++ Tools product also works with the WorkFrame/2 product (61G1177, 61G1427) as described in "IBM WorkFrame/2 Support" on page 20. It is also complemented by the tools the OS/2 2.0 Developer's Toolkit (10G3355, 10G4335) as described in "Hardware, Software, and Operating System Requirements" on page 20 and in Chapter 6, "Finishing Your Program" on page 123.

Enhanced Editor (EPM)

The Enhanced editor, referred to in this book as EPM, supports context-sensitive help for C/C++ Tools source files. It uses the files DDE4C.NDX, DDE4CPP.NDX, DDE4CCL.NDX, and DDE4UIL.NDX, provided by the C/C++ Tools product, to map each keyword to the command to open the appropriate online reference at the help panel for that keyword.

WorkFrame/2 Product

The PM-based WorkFrame/2 product is a complementary offering to the C/C++ Tools compiler and tools, and provides an adaptable, project-oriented development environment.

The C/C++ Tools product provides compiler options dialogs to be used from the WorkFrame/2 environment, and includes a number of sample projects that demonstrate the capabilities of the WorkFrame/2 and C/C++ Tools combination. An online tutorial is also provided with the WorkFrame/2 product to help you use the C/C++ Tools product in the WorkFrame/2 context.

You can easily customize the WorkFrame/2 interface to integrate your own 16-bit and 32-bit tools and create a personalized environment.

OS/2 2.0 Developer's Toolkit

The Toolkit provides header files for the OS/2 APIs and the OS2386.LIB. If your applications use any operating system services, you need these header files and library. The Toolkit also contains the LINK386 linker and NMAKE utility. In addition, it provides many tools that integrate fully into the WorkFrame/2 environment. For a short description of the Toolkit tools most commonly used with the C/C++ Tools product, see Chapter 6, "Finishing Your Program" on page 123.

Chapter 3. An Introduction to Using the C/C++ Tools Compiler

This chapter shows the basic steps for compiling, linking, and running C and C++ programs, using two of the C/C++ Tools sample programs.

PMLINES is a PM program that displays a standard window and then draws lines in the window. Both the line and background colors change. Two versions of PMLINES are shipped with the C/C++ Tools product, one written in C and one in C++. (A debugger version of the program, DPMLINES, is also included.)

Note: You must have the Toolkit installed in order to build and run the PMLINES programs. To run the C++ version, you must also have installed the User Interface class library.

Compiling a Sample C Program

The PMLINES C program, SAMPLE1A, uses C/C++ Tools library functions, OS/2 APIs, and application-defined functions. It creates and displays a standard window, uses simple menus and dialog boxes, uses a second thread, and displays graphics.

The SAMPLE1A.C source file:

- Includes the C/C++ Tools header files <**string.h**>, <**stdlib.h**>, and the Toolkit header file <**os2.h**>.
- Calls functions from the C/C++ Tools subsystem library DDE4NBS.LIB and APIs from the Toolkit library OS2386.LIB.
- Includes the user header file "pmlines.h", which defines a number of macros and identifiers.

Prototypes and defines the following application functions which all call OS/2 APIs:

InitTitle	Initializes the window title and task switch list.
ClientWindowProc	Uses a switch statement and PM APIs to handle user input, such as a request to change the line color.
DrawingThread	Draws the lines.
HelpDlgProc	Accesses the help dialog.
DisplayMessage	Displays error messages.

Creates a second thread for the DrawingThread function.

The SAMPLE1A program also makes use of a resource file, SAMPLE1A.RES, that defines the resources it uses, such as icons.

If you installed the sample programs, the C source files for SAMPLE1A are found in the SAMPLES\SAMPLE1A directory under the main installation directory. Two make files that build the sample are also provided, MAKE1AS for static linking and MAKE1AD for dynamic linking.

To compile and link SAMPLE1A, at the prompt in the SAMPLES\SAMPLE1A directory, use NMAKE with the appropriate make file. For example:

```
NMAKE all /f MAKE1AS
```

To compile and link the program yourself, use the following commands. Each step is discussed in detail in Part 2, "Compiling and Linking Your Program" on page 27.

Command	Description
icc /Rn /B"/PM:pm" SAMPLE1A.C	Compiles and links SAMPLE1A.C using the default options and the subsystem library, and passes the /PM:pm option to the linker to specify it is a PM program.
Note: The icc command invokes the linker for you. For information on linking separately from the compile step, see "Linking Independently of the Compiler" on page 123.	
rc SAMPLE1A.RES SAMPLE1A.EXE	Binds the necessary resources into the executable file.

To run the program, type SAMPLE1A at the command prompt.

Compiling a Sample C++ Program

The PMLINES C++ program, SAMPLE1B, uses primarily the User Interface class library to create and display a standard window, handle events, use a second thread, and display graphics. It also demonstrates the use of native PM APIs with User Interface class library objects.

The SAMPLE1B.CPP source file:

- Includes the Toolkit header file <**os2.h**>.
- Includes the User Interface class library header files <**iapp.hpp**>, <**ireslib.hpp**>, <**itrace.hpp**>, and <**imsgbox.hpp**>.
- Includes the user header files "pmlines.h", which defines a number of symbols and constants, and "pmlines.hpp", which contains the user class declarations.
- Constructs the main window using the application-defined MyWindow class.
- Constructs the client window using the application-defined MyClientWindow class.
- Creates a second thread for the DrawLines function, which is a member of the MyClientWindow class.
- Uses a switch statement and the User Interface class library event handling classes to handle user input, such as a request to change the line color, as well as to display messages.
- Defines the DrawLines function using direct calls to OS/2 APIs.

The SAMPLE1B.HPP header file:

- Uses preprocessor directives to ensure it is only included once.
- Includes a number of User Interface class library header files.
- Declares the class MyWindow, which is a subclass of the User Interface class library IFrameWindow class. This class declares the event handlers for the window.

Declares the class MyClientWindow, which is also a subclass of IFrameWindow. This class declares the member function DrawLines that draws the lines on the screen and also provides the functions needed to set window dimensions, window colors and to issue events to the second thread.

Like SAMPLE1A, SAMPLE1B also uses a resource file, SAMPLE1B.RES, to define the resources it needs.

If you installed the sample programs, the C++ source files for SAMPLE1B are found in the SAMPLES\SAMPLE1B directory under the main installation directory. Two make files are also provided, MAKE1BS for static linking and MAKE1BD for dynamic linking.

To compile and link SAMPLE1B, at the prompt in the SAMPLES\SAMPLE1B directory, use NMAKE with the appropriate make file. For example:

```
NMAKE all /f MAKE1BS
```

To compile and link the program yourself, use the following commands. Each step is discussed in detail in Part 2, "Compiling and Linking Your Program" on page 27.

Command	Description
icc /B"/PM:pm" SAMPLE1B.CPP	Compiles and links SAMPLE1B.C using the default options and passes the /PM:pm option to the linker to specify it is a PM program.
Note: The icc command invokes the linker for you. For information on linking separately from the compile step, see "Linking Independently of the Compiler" on page 123.	
rc SAMPLE1B.RES SAMPLE1B.EXE	Binds the necessary resources into the executable file.

To run the program, type SAMPLE1B at the command prompt.

Part 2. Compiling and Linking Your Program

This part of the Programming Guide describes the input to the compiler, how to compile and link programs, how to set compiler options, and how to use the compiler listing. It also describes static and dynamic linking of programs. In addition, it discusses some of the other Toolkit tools you may want to use to complete your application.

Compiling Your Program

Chapter 4. Compiling Your Program

The icc command invokes the C/C++ Tools compiler, which takes your C or C++ source code as input and produces an intermediate code file, a preprocessed file, or an object file. Also, the icc command by default invokes the LINK386 linker to link the object file into an executable module or a dynamic link library (DLL). This chapter describes the icc command and how to use it to compile and link your C and C++ code. It also describes how to use the intermediate code linker to improve the optimization of your code.

You can invoke icc like any other OS/2 program, such as from an OS/2 command line or using a .CMD file. You can also invoke the compiler from within a program by using the system function. For example:

```
system("icc myprog.c");
```

See the *C Library Reference* for more information about the system function.

Note: The icc command uses the LINK386 linker. To compile without linking, use the icc command with the /C+ option. Then you can link your program yourself, using either the LINK386 linker or any other linker that processes IBM 32-bit object files. See "Linking Independently of the Compiler" on page 123 for the LINK386 syntax. More information about LINK386 is provided in the Toolkit online *Tools Reference*.

If you are linking C++ files, you must invoke the linker through icc to ensure that template functions and classes are resolved correctly. You must also specify the /Tdp compiler option.

Using the icc Command

The syntax for the icc command is as follows:

```
──icc──┬─@response_file──────────────────────────────────────────┬──
       │ ┌─────────────────────────────────────────────┐         │
       │ │ ┌───────────────┐                           │         │
       └───┴─┬─────────────┼─┬───────────────────────┼─┴─┬─────────────────┬─┘
             ├─/local_optn─┤ ├─source────────────────┤   └─/global_optn────┘
             └─/global_optn┘ ├─intermediate_file─────┤
                             ├─object────────────────┤
                             ├─library───────────────┤
                             └─def_file──────────────┘
```

Depending on how you want to compile your files and how you have set up the ICC environment variable, many of the parameters used with the icc command are optional when you issue the command from a command line.

For example, to compile and link the program bob.c, you would enter the following:

```
icc bob.c
```

An object code file bob.obj, and an executable file bob.exe are created.

Local and global compiler options are discussed in "Scope of Compiler Options" on page 75.

To see an online list of all the C/C++ Tools compiler options type at the OS/2 command line:

```
icc ?
```

This listing is printed to stderr, but you can use the OS/2 redirection symbols to redirect it to stdout or to a file.

Note: The listing generated by this command is not intended to be used as a programming interface.

Compiling Programs with Multiple Source Files

To compile programs that use more than one source file, specify all the file names on the command line. For example, to compile a program with three source files (mainprog.c, subs1.c, and subs2.c), type:

```
icc mainprog.c subs1.c subs2.c
```

The source file containing the main module can be anywhere in the list. The executable module will have the same file name as the first file name but with the extension .EXE.

You can compile a combination of C and C++ files, for example:

```
icc cprog.c cppprog.cpp cxxprog.cxx othprog.oth
```

The file extension determines whether the file is compiled as a C file (.c or any other unrecognized extension) or as a C++ file (.cpp or .cxx). Given the preceding command line, cprog.c and othprog.oth are compiled as C files, and cppprog.cpp and cxxprog.cxx are compiled as C++ files.

You can also use the /Tc, /Tp, and /Td options to specify whether a file is a C or C++ file, regardless of its extension. The /Tc and /Tp options apply only to the file name immediately following the option, and specify that the file is a C file (/Tc) or a C++ file (/Tp). For example, given the following command line:

```
icc /Tc cprog.cpp cppprog.cpp /Tp cxxprog.c
```

cprog.cpp is compiled as a C file, and cppprog.cpp and cxxprog.c are compiled as C++ files.

The /Td option applies to all files that follow on the command line. Use /Tdc to specify that all source and unrecognized files that follow are to be treated as C files, or /Tdp to specify that they are to be treated as C++ files. You can specify /Td to return to the default handling of files.

For example, given the following command line:

```
icc /Tdp cxxprog.c othprog.oth /Td newprog.new
```

cxxprog.c and othprog.oth are compiled as C++ files, and newprog.new is compiled as a C file because /Td reset the default handling of files (files with unrecognized extensions are treated as C files).

Using Response Files

Instead of specifying compiler options and source files on the OS/2 command line, you can use a response file as input to icc. A response file is a flat file that contains a string of options and file names be passed to icc. The string does not specify icc itself. For example, given the command line:

```
icc /Sa /Fl catherine.c
```

the response file would contain the string:

```
/Sa /Fl catherine.c
```

The command string can span multiple lines. No continuation character is required. The string can also be longer than the limit imposed by the OS/2 command line. In some situations you may have to use a response file to accommodate a long command line, such as when you use the intermediate code linker or compile C++ code containing templates.

Because the compiler appends a space to the end of each line in the response file, be careful where you end a line. If you end a line in the middle of an option or file name, the compiler may not interpret the file as you intended. For example, given the following response file:

```
/Sa /F
l catherine.c
```

the compiler would construct the command line:

```
icc /Sa /F l catherine.c
```

The compiler would then generate an error that the /F option is not valid, and would try to compile and link the files l.obj and catherine.c.

You cannot specify another response file within the response file.

A response file can have any valid file name and extension. To use the response file, specify it on the icc command line preceded by the at sign (@). For example:

```
icc @d:\response.fil
```

No space can appear between the at sign and the file name. If you use a response file, you cannot specify other options or file names on the icc command line. Options and file names set in the ICC environment variable are still used.

Controlling Compiler Input

This section describes the methods you can use to control input to the compiler.

File Types

The C/C++ Tools compiler uses file extensions to distinguish between the different types of file it uses. The default file extensions are:

.asm	assembler listing file
.brs	browser file
.c	C source file
.cpp	C++ source file
.cxx	C++ source file
.ctn	temporary file
.def	definition file
.dll	dynamic link library
.exe	executable file
.h	header file
.hpp	header file
.i	preprocessor output file
.l	temporary file
.lst	listing file
.lib	library file
.m	temporary file
.map	map file
.obj	object file
.w	intermediate file
.wh	intermediate file
.wi	intermediate file
.wli	temporary file

For example, when you are using C/C++ Tools defaults, the command:

```
icc module1.c module2.obj mylib.lib mydef.def
```

compiles the source code file module1.c and produces the object file module1.obj. When the linker is invoked, the object files module1.obj (created during this invocation of the compiler) and module2.obj (created previously), the library file mylib.lib and the definition file mydef.def are passed to the linker. The result is an executable file called module1.exe.

OS/2 Environment Variables for Compiling

The C/C++ Tools compiler makes use of the OS/2 environment variables to provide path information and default values for compiler options. If the C/C++ Tools installation program updated your CONFIG.SYS file, many of the environment variables were set to default values for the compiler. If the program did not update CONFIG.SYS, you can set these values by running the CSETENV.CMD file in your OS/2 session before using the compiler.

Note: Some environment variables, for example ICC, are optional. They are not added to your CONFIG.SYS file or to CSETENV.CMD. If you want to use them, you can add them to either of these files and set them to the required value.

The environment variables described in this section are called the **compiler** environment variables. A number of environment variables are also used at run time. See Chapter 7, "Setting Runtime Environment Variables" on page 133 for information on the runtime environment variables.

The following OS/2 environment variables affect the operation of the C/C++ Tools compiler during compilation.

PATH

The PATH variable names the directory (or directories, separated by semicolons) searched for executable modules when the compiler is invoked. This variable should include the directories containing the C/C++ Tools compiler and LINK386 executable modules.

DPATH
The compiler searches for help and message files in the directories specified by this variable.

INCLUDE
The compiler searches for the header files in the directories listed by this variable.

LIB
This variable should include the directories containing the C/C++ Tools libraries (.LIB files) that are used by the linker. If you call any OS/2 APIs, OS2386.LIB from the Toolkit must also be specified by the LIB variable.

TMP
This variable contains the path where the C/C++ Tools compiler places all its temporary work files. If this variable is undefined, the compiler uses the current directory. If you installed the compiler on a LAN, temporary files are stored in your local directory. The work files created by the compiler are normally erased at the end of compilation; however, if an interruption occurs during compiling, these work files may still exist after the compilation ends. If you set the TMP variable, you eliminate the possibility of work files being scattered around your file system. See also Chapter 5, "Using Compiler Options" on page 71 for information on the /Fd compiler option, which you can use to control whether temporary files are stored in shared memory or on disk. (Note that if you are compiling a C++ program, you must store temporary files on disk.)

Setting TMP to point to a virtual disk (also called a RAM disk) may improve compilation time. See the OS/2 documentation for information on using the VDISK device driver to create a virtual disk.

ICC
You can use this variable to specify default compiler options as well as source file names. See "Specifying Compiler Options" on page 71 for a more detailed description of the ICC variable.

Setting Environment Variables

Environment variables are given values using the OS/2 SET command. The LIBPATH variable and all DEVICE statements must be set in CONFIG.SYS. For all other variables, you can use the SET command in three places:

CONFIG.SYS file
You can add the environment variables to the CONFIG.SYS file. If the environment variable already exists in CONFIG.SYS, add the C/C++ Tools values to the existing variable. You can also have the C/C++ Tools installation program update CONFIG.SYS for you.

Environment variables specified in CONFIG.SYS are in effect for every session you start. This is a good place to specify variables that you want to apply each time you compile.

CSETENV.CMD file
This is an OS/2 command file that is created by the C/C++ Tools installation program. You must use this command file each time you start a session in which you are going to use the C/C++ Tools product. The variables are in effect only for the session in which you use the CSETENV.CMD file.

If you use the installation defaults (that do not update CONFIG.SYS), CSETENV.CMD contains the following statements:

```
@REM  DEVICE=C:\IBMCPP\SYS\DDE4XTRA.SYS
@REM  LIBPATH=C:\IBMCPP\DLL;
SET  PATH=C:\IBMCPP\BIN;%PATH%
SET  DPATH=C:\IBMCPP\LOCALE;C:\IBMCPP\HELP;C:\IBMCPP\SYS;%DPATH%
SET  LIB=C:\IBMCPP\LIB;%LIB%
SET  INCLUDE=C:\IBMCPP\INCLUDE;C:\IBMCPP\IBMCLASS;%INCLUDE%
SET  HELP=C:\IBMCPP\HELP;%HELP%
SET  BOOKSHELF=C:\IBMCPP\HELP;%BOOKSHELF%
SET  HELPNDX=DDE4C.NDX+DDE4CPP.NDX+DDE4UIL.NDX+DDE4CCL.NDX+%HELPNDX%;
SET  TMP=C:\IBMCPP\TMP
SET  TZ=EST5EDT, , , , , , , ,
```

Adding environment variables of your choice to this file is a way of specifying variables that you always use without having to type them individually on the command line. The variables in this file override environment variables in the CONFIG.SYS file. You can append the original value of a variable using *%variable%*, as shown in this PATH statement:

```
SET PATH=C:\IBMCPP\BIN;%PATH%
```

command line

When the SET command is used on the OS/2 command line, the values you specify are in effect only for that OS/2 session. They override values previously specified in CONFIG.SYS or by CSETENV.CMD. You can append the original value of a variable using *%variable%*, as shown above in the example for PATH.

Example of Setting Environment Variables

The following example could be used in the CSETENV.CMD file or on the command line.

If the executable files that make up the compiler are in C:\IBMCPP\BIN, the following command adds this directory to the PATH variable:

```
SET PATH=C:\IBMCPP\BIN;%PATH%
```

This command makes C:\IBMCPP\BIN the first directory searched by the OS/2 operating system (after the current directory). To put the new directory at the end of the search sequence, put %PATH% before the new directory name.

Source File Names in ICC

In addition to compiler options, you can also put file names into the ICC variable. For example, if you specify:

```
SET ICC=test.c check.c
```

the command

```
icc main.c
```

causes test.c, check.c, and main.c to be compiled and linked, in that order. You can also specify intermediate files (created with the /Fw option) in ICC. They are treated like source files.

All the library, object, or module definition files that appear in ICC will be passed to the linker when it is invoked.

Controlling #include Search Paths

The #include preprocessor directive allows you to retrieve source statements from secondary input files and incorporate them into your program.

You can nest #include directives in an included file. There is a limit of 128 levels of nesting in the C/C++ Tools compiler.

Compiler options and environment variables let you choose the disk directories searched by the compiler when it looks for #include files.

This section describes how to specify #include file names and how to set up search paths for these files.

#include Syntax

```
───#include──┬──<filename>──┬──────────────
             └──"filename"──┘
```

In the above figure, angle brackets indicate a **system** #include file, and quotation marks indicate a **user** #include file.

#include File Name Syntax

You can specify any valid OS/2 file name in a #include directive. The file name must be sufficiently qualified for the compiler to be able to locate the file. In some cases, an unqualified or partially qualified file name may be sufficient; in others, you may have to include the entire path name.

If a path name is too long to fit on one line, you can place a continuation character, or backslash (\), at the end of the unfinished line to indicate that the current line continues onto the next line. The backslash can follow or precede a directory separator, or divide a name. For example, to include the following file as a user #include file:

```
c:\cset\include\mystuff\subdir1\subdir2\subdir3\myfile.h
```

You could insert one of the following #include directives in your program:

```
#include "c:\cset\include\mystuff\subdir1\sub\
dir2\subdir3\myfile.h"
```

or

```
#include "c:\cset\include\mystuff\subdir1\\
subdir2\subdir3\myfile.h"
```

Notes:

1. The continuation character must be the last non-white-space character on the line. (White space includes any of the space, tab, new-line, or form-feed characters.) The line cannot contain a comment.
2. The continuation character, although the same character as the directory separator, does not take the place of a directory separator or imply a new directory.

Ways to Control the #include Search Paths

You can control the #include search paths in three ways:

- Use the /I, /Xc, and /Xi compiler options on the command line when invoking the compiler.
- Use the /I, /Xc, and /Xi compiler options in the ICC environment variable.
- Specify the search paths in the INCLUDE environment variable.

For more information on the compiler options /I, /Xc, and /Xi, see "#include File Search Options" on page 86.

#include Search Order

When the compiler encounters either a user or system #include file statement with a fully-qualified file name, it looks only in the directory specified by the name.

When the compiler encounters a user #include file specification that is not fully qualified, it searches for the file in the following places in the order given:

1. The directory where the original top-level file was found.
2. Any directories specified using /I that have not been removed through the use of /Xc. Directories specified in the ICC environment variable are searched before those specified on the command line.
3. Any directories specified using the INCLUDE environment variable, provided that the /Xi option is not currently in effect.

When the compiler encounters a system #include file specification that is not fully qualified, it searches for the file in the following places in the order given:

1. Any directories specified using /I that have not been removed through the use of /Xc. Directories specified in the ICC environment variable are searched before those specified in command line.
2. Any directories specified using the INCLUDE environment variable, provided that the /Xi option is not currently in effect.

Accumulation of Options

The #include search options are cumulative between the ICC and INCLUDE environment variables and the command line. For example, given the following ICC and INCLUDE environment variables

```
ICC=/I\rosanne
INCLUDE=c:\kent;\alan
```

and the following command line

```
icc /Xi+ /Ic:\connie test.c /Xi- /Xc /Id:\dal f:\moe\marko\jay.c
```

any system #include files referenced in the file test.c will be searched for first in the directory \rosanne and then in the directory c:\connie. Because the /Xi+ option was specified, none of the directories set in the INCLUDE environment variable will be searched.

Using the same example, any user #include files referenced in test.c would be searched for first in the current directory, then in the directory \rosanne, and then in c:\connie.

Any system #include files referenced in the file f:\moe\marko\jay.c will be searched for first in the d:\dal directory, then in the c:\kent directory, and finally the \alan directory. The directories specified in the INCLUDE variable are searched because the /Xi- option overrides the /Xi+ option specified previously. The /Xc option removes the directories \rosanne and c:\connie from the current search path.

Any user #include files referenced in jay.c will be searched for in the following directories, in the given order: f:\moe\marko, d:\dal, c:\kent, and \alan.

Setting the Source Code Language Level

You can set the language level of your source code to one of four language levels, which are described below. You can set the level using compiler options either on the command line or in ICC, or by using a #pragma langlvl directive. Note that a #pragma langlvl directive set in your source code overrides any language-level compiler options specified on the command line or in ICC. When you set the language level, you also define the macro associated with that level.

Setting the Language Level

The language levels are:

1. **ANSI** - Allow only language constructs that conform to ANSI C standards or for C++ code, that conform to the standards in the ANSI working paper on C++ standards. All non-ANSI constructs cause compiler errors.

   Use this language level to write code that is portable across ANSI-conforming systems.

   To allow only ANSI constructs, use the /Sa option or #pragma langlvl(ansi), which define the macro __ANSI__.

2. **SAA Level 2** - Allow only language constructs that conform to SAA Level 2 C standards. This language level is valid for C code only, because there is no SAA standard for C++. SAA constructs include all those allowed under the ANSI language level, because the SAA C standard conforms to the ANSI standard. All non-SAA constructs cause compiler errors. See the *SAA CPI C Reference - Level 2* for a full description of the SAA C standard.

   Use this language level to write code that is portable across SAA systems.

   To allow only SAA constructs, use the /S2 option or #pragma langlvl(saal2), which define the macro __SAA_L2__.

   Note: You can also use #pragma langlvl(saa), which defines the macro __SAA__. This level allows constructs that conform to the most recent SAA C definition. Because Level 2 is currently the most recent definition, the __SAA__ and __SAA_L2__ macros are equivalent at this time.

3. **Extended** - Allow all C/C++ Tools language constructs. These include all constructs that fall under the ANSI and SAA Level 2 language levels and the C/C++ Tools extensions to those standards.

   This is the default language level.

   To explicitly state this default (for example, on the command line to override a setting in ICC), use the /Se option or #pragma langlvl(extended), which define the macro __EXTENDED__.

4. **Compatible** - Allow constructs and expressions that were allowed by earlier levels of the C++ language. This language level is valid for C++ code only.

   When the language level is set to compatible:

   - Classes declared or defined within classes or declared within argument lists are given the scope of the closest non-class.
   - typedefs and enumerated types declared within a class are given the scope of the closest non-class.
   - The overload keyword is recognized and ignored.
   - An expression showing the dimension in a delete expression is parsed and ignored. For example, given:

     ```
     delete [2 ] p;
     ```

     2 is ignored.
   - Conversions from const void and volatile void to void are allowed. At other language levels, these conversions would require an explicit cast.
   - Where a conversion to a reference type uses a compiler temporary type, the reference need not be to a const type.
   - You can bypass initializations as long as they are not constructor initializations.
   - You can return a void expression from a function that returns void.
   - operator++ and operator-- without the second zero argument are matched with both prefix and postfix ++ and --.
   - You can use the $ character in identifiers. Note that you can also use $ in C++ files when the language level is set to extended.
   - In a cast expression, the type to which you are casting can include a storage class specifier, function-type specifier (inline or virtual), template specifier, or typedef. At other language levels, the type must be a data type, class, or enumerated type.
   - You can have a trailing comma in a list of enumerators, for example, enum E {e , };.
   - Given the expression class A a = new(x) A[1];, the compiler looks for a member operator new because the placement syntax (new(x)) is used. The member operators are not typically used to allocate arrays.
   - You can use the comma operator.

- You can declare a member function using both the inline and static keywords, for example, inline static void sandra :: pete(void);. The static keyword is ignored.
- No error is generated if a function declared to return a non-void type does not contain at least one return statement. Such a function can also contain return statements with no value without generating an error.
- If two pointers to functions differ only in their linkage types, they are considered to be compatible types.

Use this language level to write code that is portable to systems with older implementations of C++, or to port older code to the C/C++ Tools product.

To allow older C++ constructs, use the /Sc option or #pragma langlvl(compat), which define the macro __COMPAT__.

If you want to write portable C code to be compiled and run on different SAA platforms, you should use the /S2 option. The SAA C standards conform to the ANSI standards, but also feature some additional elements. If you will be compiling and running your code primarily on the personal computer platform, you should use the /Se option. If you will be compiling and running your code on other non-SAA platforms, you should use the /Sa option.

Controlling Compiler Output

The icc program can generate the following output:

- An object module for each C/C++ source file input.
- One executable module (or dynamic link library).
- A listing file for each C/C++ source file that contains information about the compilation.
- Preprocessed header files.
- Template-include files. See "Generating Template-Include Files" on page 231 for more information about these files.
- A linker map file.

- A preprocessor output file for each C/C++ source file. You can use this output file for debugging information.

  Note: This information is not intended to be used as a programming interface.

- An assembler listing file for each C/C++ source file. The format of the listing is in the style of the MASM 5.1 assembler output. The C/C++ source is annotated in the listing. Assembler listings will not always compile, especially if reserved MASM keywords are used as external variables or functions.

  Note: This listing is not intended to be used as a programming interface.

- A browser listing file for use by the C/C++ Tools browser.

- Intermediate code files. Three files (.w, .wh, .wi) are produced per source file.

  Note: These files are not intended to be used as a programming interface.

- Temporary files.

  Note: These files are not intended to be used as a programming interface.

- Diagnostic information about possible programming errors.

  Note: This information is not intended to be used as a programming interface.

- Messages (for example, the IBM logo and help messages).

Object Files

The object files that are produced by the C/C++ Tools compiler can be linked to create either executable (.EXE) files or dynamic link libraries (.DLL files). Use the /Ge+ option to create an executable file or /Ge- to create a DLL. See "Code Generation Options" on page 111 for more information on using compiler options to specify the type of object file to be created.

Optimization Level of Object Code

The C/C++ Tools compiler can perform many optimizations, such as local optimizations, common subexpression elimination, and loop optimizations on object code. Use the /O+ option to enable optimization. By default, optimization is turned off (/O-). You can also control the inlining of user code with the /Oi option, the use of the intermediate code linker with the /Ol option, the inclusion of optimizations that involve the instruction pointer with the /Op option, and the use of the instruction scheduler with the /Os option.

See "Code Generation Options" on page 111 for more information on using compiler options to control optimization. For more information on how you can optimize your code, see Chapter 10, "Optimizing Your Program" on page 165.

Generating Debugger Information

The information necessary for running the C/C++ Tools debugger can be placed in the object file produced by the compiler using the /Ti+ option. To include the debugger information in the executable file or DLL, use the /DE linker option. If you use icc to invoke the linker and specify /Ti+, the /DE option is automatically passed to the linker.

When you use /Ti+, do not turn on optimization (/O+, /Oi+, or /Os+). Because the compiler produces debugging information as if the code were not optimized, the information may not accurately describe an optimized program being debugged, and the debugger will not operate properly.

Because of the effects of optimization, debugging information generated with optimization is limited to setting breakpoints at function entry and function exit and stepping through the program at assembly level. Accurate symbol and type information is not always available.

To make full use of the C/C++ Tools debugger, set optimization off and use the /G3 option. (Note that these are the defaults.)

See "Debugging and Diagnostic Information Options" on page 92 for more information on using compiler options to control the generation of debugging information.

Generating EXTRA Information

To include the information required by EXTRA in the object file, use both the /Ti+ and /Gh+ options. To include the EXTRA information in the executable file or DLL, use the /DE linker option. If you use icc to invoke the linker and specify /Ti+, the /DE option is automatically passed to the linker.

When /Gh+ is specified, the compiler generates a call to a profiling hook function as the first instruction in the prolog of each function. There are two profiling hook functions:

_ProfileHook32 Profile hook for all 32-bit functions.

_ProfileHook16 Profile hook for all 16-bit callback functions. These functions are defined with either the _Far16 _Cdecl or _Far16 _Pascal linkage keywords.

Other profiler vendors who plan to support the C/C++ Tools product must provide their own profiling hook functions to gather all necessary runtime information .

Executable Files

By default, the compiler generates one executable file for each compiler invocation. If you specify /C+, the compiler generates an object file that you can then link separately to create an executable file.

There are two types of executable files:

- Those that run in the C/C++ Tools runtime environment.

  This is the default, and most C and C++ applications run under this environment. It supports all the C/C++ Tools runtime functions and automatically provides initialization, exception management, and termination routines for C and C++.

- Those that run as subsystems.

  Programs developed as subsystems can only make use of a subset of the C/C++ Tools runtime library. You have to take care of initialization, exception management, and termination using OS/2 services and APIs.

Subsystems are intended for developing applications that cannot have a resident environment, such as Presentation Manager display and printer drivers. If your application does not require the C/C++ Tools runtime environment, you can also use the subsystem library to reduce your program's size and improve its performance. To compile a subsystem executable file, use the /Rn option.

For more information on subsystems and their uses, see Chapter 17, “Developing Subsystems” on page 303. For information on the compiler options used to produce subsystems see “Code Generation Options” on page 111.

You can use a number of compiler options to change the executable file created by the compiler. See “Code Generation Options” on page 111 for information on using compiler options to specify the type of executable file you want to create.

Compiler Listings

When you compile a program, you can produce a listing file that contains information about the source program and the compilation. You can use this listing to help you debug your programs.

Note: The compiler listing file is not intended to be used as a programming interface.

At the very minimum, the listing will show the options used by the compiler, any error messages, and a standard header that shows:

- The product number
- The compiler version and release number
- The date and time compilation commenced
- A list of the compiler options in effect.

For information on how to use compiler options to specify the information and format of this file, see “Listing File Options” on page 88.

Temporary Files

The C/C++ Tools compiler creates and uses temporary files during compilation. Temporary files are usually erased at the end of a successful compilation.

When you compile C++ files, the temporary files are stored on disk. When you compile C files, they can be stored either on disk or in shared memory. Although the compiler runs faster using shared memory, if you do not have a lot of memory available (for example, you have many OS/2 programs running or very little memory installed), there is no benefit in using shared memory. The time saved by using shared memory over disk may only be apparent for compilation of large programs.

To specify that the temporary files are stored on disk rather than in shared memory, use the following command:

```
icc /Fd+ myprog.c
```

The temporary files created by the compiler are erased at the successful end of compilation; however, if the compilation is interrupted, these files may be left on the disk. They are located in the path specified by the TMP environment variable. If you use memory files and they overflow to the disk, they will also be located in the path specified by TMP. If this variable is undefined, the compiler uses the current directory. For more information on the TMP variable, see “OS/2 Environment Variables for Compiling” on page 34 and Chapter 7, “Setting Runtime Environment Variables” on page 133.

Compilation time may be improved if you specify a virtual disk as the location for the temporary files. Copying the compiler files from the BIN directory onto a virtual disk can also improve compilation time. See the OS/2 documentation for information on using the VDISK device driver to create a virtual disk.

Messages

You can use compiler options to control:

1. The level of error message that the compiler outputs and that increments the error count maintained by the compiler (with the /W*n* option).
2. How many errors are allowed before the compiler stops compiling (with the /N*n* option).
3. The diagnostics run against the code (with the /W*grp* option).

See "Debugging and Diagnostic Information Options" on page 92 for more information on using the compiler options to control messages.

Return Codes

The C/C++ Tools compiler returns the highest return code it receives from executing the various phases of compilation. These codes are:

Code	Meaning
0	The compilation was completed, and no errors were detected. Any warnings have been written to stdout. Your executable file should run successfully.
12	Error detected; compilation may have been completed; successful execution impossible.
16	Severe error detected; compilation terminated abnormally; successful execution impossible.
20	Unrecoverable error detected; compilation terminated abnormally and abruptly; successful execution impossible. If the error code is greater than 20, contact your IBM service representative.

For every compilation, the compiler generates a return code that indicates to the operating system the degree of success or failure it achieved.

Precompiled Header Files

You can use the /Fi+ compiler option to create or recreate precompiled versions of every source header file used during that compilation.

The precompiled version of each header file is created in a subdirectory called CSET2PRE under the directory containing the original header file. For example, the precompiled version of d:\brolley\luc.h is placed in the directory d:\brolley\CSET2PRE. If the subdirectory does not exist, the compiler creates it for you. The file name given the precompiled header file is the same as the original name. The timestamp is also the same as that of the original file so the compiler can ensure that the precompiled file is current.

To use the precompiled header files, specify the /Si+ option. For each #include statement, the compiler determines which header file is required using the usual #include search path. (See “#include Search Order” on page 40 for more information on the #include search path.) It then looks for the precompiled version in the CSET2PRE subdirectory under the directory containing the original header file. It uses the precompiled version if it exists and is current. For example, given:

d:\brolley\local contains luc.h and emily.h
d:\brolley\temp contains emily.h
The search path is d:\brolley\temp;d:\brolley\local
The source file contains the statements:

```
#include "luc.h"
#include "emily.h"
```

the compiler looks for the precompiled header files as follows:

luc.h in the directory d:\brolley\local\CSET2PRE
emily.h in the directory d:\brolley\temp\CSET2PRE.

Note that a precompiled header file is only used if the original header file exists. If the original file has been deleted, renamed, or moved to another directory, the precompiled version is **not** used. For example, if you erase d:\brolley\temp\emily.h, the compiler does not use the precompiled header file in d:\brolley\temp\CSET2PRE. Instead it continues along the search path, finds d:\brolley\local\emily.h, and uses the precompiled header file for this file if it exists and is current.

You can use /Fi+ and /Si+ together to automatically create and maintain precompiled header files for your application. If you use the options consistently, precompiled header files are created if they do not exist, and used if they do. When a source file is changed, the precompiled version is automatically regenerated.

You can create a precompiled header file when you compile a C program and use it when you compile C++ code, providing the content of the header file is valid for both languages. The converse is also true. For more information on writing header files that can be used for both C and C++, see the appendix on C – C++ Compatibility in the *C++ Language Reference*.

When you use precompiled header files, keep the following restrictions in mind:

- To create a precompiled header file, the compiler process must have write permission to the CSET2PRE subdirectory or permission to create the subdirectory if it does not exist. To use a precompiled header, the compiler process must have read permission for that file.
- Precompiled header files do not appear in any listing files.
- If you specify /P+ to run the preprocessor only, the /Fi and /Si options are ignored.

Using the Intermediate Code Linker

The intermediate code linker combines the information in all .w, .wh, and .wi intermediate code files into one set of files that is then used by the compiler to optimize the code and generate an object module.

In addition to the optimizations performed by the intermediate linker itself, using this linker exposes more of your program to the optimizer at a time. The optimizer can then generate more efficient code. Using the intermediate linker can result in improved code optimization, especially where inlining is used, and better program performance. Note that using the intermediate linker on code being compiled into an executable file results in better performance improvements than if the same code were being compiled into a DLL or library.

Using the Intermediate Code Linker

To use the intermediate linker, specify the /Ol+ option on the icc command line. For best results, use the /Gu option, as described in "Using the /Gu Option" on page 54,and specify /O+ to turn optimization on.

Note: Because optimization limits the generation of debugging information, use /O- if you want to debug your program. The /Ol option does not affect debugging information.

Given the following command:

```
icc /O+ /Ol+ vij.c thomas.c tim.c
```

the compiler:

1. Compiles each source file into a set of intermediate code files (.w, .wh, and .wi files).
2. Invokes the intermediate code linker to link the intermediate code files of all three source files.
3. Optimizes the code.
4. Creates **one** object module for all three files and names it after the first file specified on the command line (vij.obj). (You can change the name of the object file using the /Fo option.)
5. Invokes LINK386 to create an executable module (vij.exe). (You can change the name of the executable file using the /Fe option.) If you want to link your object files separately, use the /C+ option on the icc command line. You can then invoke LINK386 as you would for any other object file.

Alternatively, you can use the /Fw+ option to create and save the intermediate code files to be linked by the intermediate linker at a later time. When you use /Fw+, compilation stops when the intermediate files are created. For example:

```
icc /Fw+ brian.c jim.c
```

creates only the intermediate files for brian.c and jim.c. No object or executable modules are created.

The /Fw option also takes an optional file-name parameter that lets you specify the file name for the intermediate files. For example:

```
icc /Fwtony jeff.c
```

names the resulting intermediate files for jeff.c to tony.w, tony.wh, and tony.wi. Note that there is no space allowed between /Fw and the file-name parameter.

You can specify existing intermediate files on the icc command line to run the intermediate linker and complete the compilation. You need only specify the name of the .w file; the .wh and .wi files are included automatically. No option is required. For example, the command:

```
icc brian.w jim.w
```

links all intermediate files for brian.c and jim.c, creates an object file, and invokes LINK386 to create an executable module.

Note: You cannot use compiler options related to source files with intermediate files because the source has already been partially compiled. For example, you cannot produce a listing file from intermediate files or set the language level for the program.

You can also combine intermediate and source files on the command line to run the intermediate linker on all the files and complete the compilation. No option is required. For example:

```
icc brian.w jim.c
```

Note: If you use the intermediate code linker on a large application, you will require more system resources than if you were simply compiling. For example, compiling and intermediate linking a 40000-line application requires a working set of approximately 25M. If your executable module or DLL contains more than 100000 lines of code, using the intermediate code linker is not recommended.

Using the /Gu Option

One of the optimizations performed by the intermediate linker is to discard any defined data or functions that are:

- Not referenced in the files included in the link.
- Not defined as exports either by the _Export keyword, by #pragma export, or in the .DEF file. (Note: If you define exports in the .DEF file, you must include the file name in the icc command line.)

If you call functions in files not included in the intermediate link, such as library functions or OS/2 APIs, this optimization cannot be performed because the data and functions could possibly be used by one of these external functions. Because library functions and APIs rarely use data defined in user code, the result is often poorly optimized code.

To ensure that all unreferenced data and functions are discarded, use the /Gu+ option. This option tells the intermediate linker that any external functions that are referenced will not use anything defined in the files being linked. Use the _Export keyword to mark any definitions that will be used in a separate compilation unit.

In addition, /Gu+ causes all external functions and data that are not exported to be defined as static, which can result in better optimization.

Error Checking

Another benefit of using the intermediate code linker is enhanced error checking of all files included in the intermediate link step. The intermediate linker can find errors that would otherwise generate linker errors or unexpected runtime behavior, such as:

- Redefinition of variables and functions
- Inconsistent declarations or definitions of the same function (including differences in return type, linkage, number of arguments, and argument properties)
- Type mismatches between different declarations or definitions of the same variable, with the exception of:
  - Differences in integer type of the same length (int and long)
  - Some mismatches within structures and unions
  - Mismatches between array declarations where one of the declarations is an external reference.

Inlining User Code

By default, the compiler inlines certain library functions, meaning that it replaces the function call with the actual code for the function at the point where the call was made. These library functions are called intrinsic or built-in functions.

You can also request that the compiler inline the code for your own functions. There are two ways to inline user code:

1. Using the _Inline keyword to specify which functions you want to have inlined. You must specify the /Oi option to turn inlining on.

   The C++ language provides the function specifier inline that you can use in the same manner as _Inline. The _Inline keyword is not supported for use in C++ programs.

2. Using the /Oi option with a *value* parameter to automatically inline functions smaller than the value specified.

Note: Requesting that a function be inlined makes it a candidate for inlining but does not necessarily mean that the function will be inlined. In all cases, whether a function is actually inlined is up to the compiler.

Using Keywords

For C files, use the _Inline keyword to qualify either the prototype or definition of the functions you want to have inlined. For example:

```
_Inline int james(int a);
```

specifies that you want james to be inlined.

In C++ files, use the inline function specifier in the same way as _Inline. For example:

```
inline int angelique(char c);
```

declares angelique is to be inlined.

The _Inline and inline keywords hav3 the same meaning and syntax as the storage class static. When you turn inlining on (with the /Oi+ or /Oi*value* option), the keywords have the added meaning of causing the function they qualify to be inlined. In addition, C++ member functions that are defined in a class declaration are considered candidates for inlining by the compiler.

Using the /Oi Option

The /Oi option controls whether user functions are inlined or invoked through a function call:

/Oi- No user code is inlined.

/Oi+ Functions qualified with the _Inline or inline keyword are inlined.

/Oi*value* Functions qualified with the _Inline or inline keyword are inlined, as are all other functions that are less than or equal to *value* in abstract code units (ACUs) as measured by the compiler. This option is called auto-inlining.

The default is /Oi-. When optimization is turned on (/O+), the default becomes /Oi+.

Note: The /Oi option does **not** affect the inlining of intrinsic C/C++ Tools library functions. To disable the inlining of library functions, parenthesize the function call, for example:

```
(strcpy)(str1, str2);
```

You cannot disable inlining for user functions, meaning you cannot request that specific functions **not** be inlined. In addition, some library functions are implemented as built-in functions, meaning there is no backing code in the library. You cannot parenthesize calls to these functions.

See the *C Library Reference* for a list of all the intrinsic and built-in library functions.

In general, choosing the functions you want inlined yields better results than auto-inlining.

Inlining User Code

If you use auto-inlining, *value* has a range between 0 and 65535 ACUs. The number of ACUs that comprise a function is proportional to the size and complexity of the function. For example, the following function is 33 ACUs:

```
int florence(char a, int b)
{
    if(a != 1 )
        b++;
    else
        b += 1 ;
    return(a);
}
```

The next function is 51 ACUs:

```
int sanjay(long par1, long par2)
{
    while(par1)
    {
        if(par2)
            test3();
        par1--;
    }

    if(par1)
        testing();
    par1 += par2;
}
```

Messages are generated to tell you which functions are inlined based on the *value* you specified. You can then adjust the *value* if necessary. Messages are not generated for functions qualified with _Inline or inline, or for C++ functions defined in a class declaration.

When you turn inlining on for C programs, small functions (of 50 ACUs or less) of static storage class that are called only once are also inlined. They are not inlined for C++ programs. You can use /Oi*value* with a very small value to display the names of these functions.

Note: The *value* required to inline a specific function may be slightly larger when /O+ is specified than when /O- is specified.

Benefits of Inlining

Inlining user code eliminates the overhead of the function call and linkage, and also exposes the function's code to the optimizer, resulting in faster code performance. Inlining produces the best results when:

- The overhead for the function is nontrivial, for example, when functions are called within nested loops.
- The inlined function provides additional opportunities for optimization, such as when constant arguments are used.

For example, given the following function:

```
void glen(int a, int b)
{
    if (a == 1 )
    {
        switch(b)
        {
            case  1:   .
                       :
             case 2 : puts("b is 2 ");
                       break;
             case 3 : .
                       :
            default:  .
                       :
        }
    }
}
```

and assuming your program calls glen several times with constant arguments, for example, glen(1 , 2);, each call to glen causes the if and switch expressions to be evaluated. If glen is inlined, the compiler can then optimize the function. The evaluation of the if and switch statements can be done at compile time and the function code can then be reduced to only the puts statement from case 2 .

The best candidates for inlining are small functions that are called often. Use EXTRA or a profiler to determine which functions to inline to obtain the best results.

To improve performance further:

- Use constant arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- If you have a function that is called many times from a few functions, but infrequently from others, create a copy of the function with a different name and inline it only in the functions that call it often.
- Turn optimization on.

Drawbacks of Inlining

Inlining user code usually results in a larger executable module because the code for the function is included at each call site. Because of the extra optimizations that can be performed, the difference in size may be less than the size of the function multiplied by the number of calls.

Inlining can also result in slower program performance, especially if you use auto-inlining. Because auto-inlining looks only at the number of ACUs for a function, the functions that are inlined are not always the best candidates for inlining. As much as possible, use the _Inline or inline keyword to choose the functions to be inlined.

When you use inlining, you need more stack space. When a function is called, its local storage is allocated at the time of the call and freed when it returns to the calling function. If that same function is inlined, its storage is allocated when the function that calls it is entered, and is not freed until that calling function ends. Ensure that you have enough stack space for the local storage of the inlined functions.

Restrictions on Inlining

The following restrictions apply to inlining:

- You cannot inline functions that use a variable number of arguments.
- You cannot inline functions with _System linkage that make use of the __parmdwords function.
- For C++, you cannot declare a function as inline after it has been called.

To use _Inline or inline, the code for the function to be inlined must be in the same source file as the call to the function. To inline across source files you must either:

1. Place the function definition (qualified with _Inline) in a header file that is included by all source files where the function is to be inlined.
2. Use the intermediate code linker (with the /Ol+ option) and auto-inlining. The intermediate code linker is described in "Using the Intermediate Code Linker" on page 52.

If you plan to debug your executable module, use /Oi- to turn off inlining. Inlining can make debugging difficult; for example, if you set a breakpoint at the entry of a function that is inlined, the breakpoint is not set at the point where the function is inlined in another function.

EXTRA treats an inlined function as part of the function in which it is inlined.

A function is not inlined during an inline expansion of itself. For a function that is directly recursive, the call to the function from within itself is not inlined. For example, given three functions to be inlined, A, B, and C, where:

- A calls B
- B calls C
- C calls back to B

the following inlining takes place:

- The call to B from A is inlined.
- The call to C from B is inlined.
- The call to B from C is not inlined because it is made from within an inline expansion of B itself.

Setting the Calling Convention

The C/C++ Tools compiler supports four 32-bit calling conventions, and three 16-bit conventions:

32-bit: _Optlink
_System
_Pascal
_Far32 _Pascal

16-bit: _Far16 _Cdecl
_Far16 _Pascal
_Far16 _Fastcall

Note: The _Far32 _Pascal convention can only be used in C programs and only when the /Gr+ option is specified.

The default is _Optlink for calls to 32-bit code. You must explicitly specify a calling convention for all 16-bit calls. If you specify only _Far16, the convention defaults to _Far16 _Cdecl. You can change the default for 32-bit code to _System by using the /Ms compiler option. The /Mp option explicitly sets the calling convention to _Optlink. See "Code Generation Options" on page 111 for more information on these compiler options.

You can also set the calling convention for individual functions using either linkage keywords or, for C programs only, the #pragma linkage directive.

For example, to declare kathryn as a function with the _System calling convention, you could use the following statement using a linkage keyword:

```
int _System kathryn(int i);
```

or in a C program, the following #pragma directive:

```
#pragma linkage(kathryn, system)
```

Note that, when using the #pragma linkage directive, you must declare the function separately. Using linkage keywords is generally quicker and easier than using #pragma linkage directives.

Both the keywords and the #pragma linkage directive take precedence over the compiler option used. If both methods are used and specify different conventions for the same function, an error message is generated.

Note: You cannot change the calling convention for C++ member functions. Member functions always use the _Optlink convention.

The linkage keywords and #pragma linkage directive are described in more detail in the *Online Language Reference*. For more information on the calling conventions and how they work, see Chapter 14, "Calling Conventions" on page 237.

Choosing Your Runtime Libraries

You can use compiler options to control the linking process by changing the type of runtime library you link to. If you do not specify any options, the compiler uses the library that produces single-thread executable modules that are statically linked. You can link to another library by specifying the appropriate options. You would link to another library to:

- Dynamically link your program (discussed in the following section).
- Create a multithread executable module. (See Chapter 11, "Creating Multithread Programs" on page 179.)
- Develop a subsystem. (See Chapter 17, "Developing Subsystems" on page 303.)
- Create a DLL for use with another executable module. (See Chapter 12, "Building Dynamic Link Libraries" on page 195.)

The naming conventions used for the libraries are intended to help identify their function. The libraries are named as follows:

Character Position	Significance
1234 5 6 7 8	
DDE4	Product prefix
S M N	Single-thread library Multithread library Subsystem library (no runtime environment)
B	Builds both executables and DLLs
S	Standard library
I O	Import library Object library (contains initialization routines) Statically bound library (no eighth letter)

For example, the library DDE4SBS.LIB is the standard single-thread library for building both executable modules and DLLs, while DDE4NBSI.LIB is the standard import library for creating a subsystem.

For a list of all libraries and files shipped with the C/C++ Tools product, see Appendix E, "Component Files" on page 431.

Static and Dynamic Linking

Static linking means that code for all the runtime functions called in your program is linked with your program in the executable module or DLL. The .EXE or .DLL files will be larger because there is a copy of the runtime functions in each file. These programs will take up more storage, and if you run them at the same time, there will also be a copy of the library functions in memory for each program. Statically-linked programs, however, are easier to distribute because the library functions are part of the executable module. See Note 1 below.

Dynamic linking does not include the actual code for the library functions in the .EXE or .DLL file. The full code for the library function is resolved at load time and the amount of disk space required by your executable modules is reduced.

You need to link to the appropriate runtime library for the kind of linking you are doing. The compiler option used to control whether your module links to the runtime library statically or dynamically is /Gd.

The default is /Gd-, which statically links the runtime library in the executable module. Static linking uses the static version of the runtime library.

To dynamically link the runtime library in an executable file, specify the /Gd+ option. The correct version of the runtime library will be dynamically linked to your executable module.

Under the C/C++ Tools licensing agreement, you cannot ship the C/C++ Tools DLLs with a product that you develop. If you do not want to statically link to the C/C++ Tools library, you can create your own version of the C/C++ Tools runtime DLLs, as described in "Creating Your Own Runtime Library DLLs" on page 216.

Notes:

1. When you use static linking, all external names beginning with Dos, Vio, or Kbd (exactly as shown) become reserved external identifiers. They are not reserved if you use dynamic linking.
2. You can also link dynamically to your own DLLs. Creating and using your own DLLs is discussed in Chapter 12, "Building Dynamic Link Libraries" on page 195.
3. When you use the /Gd+ compiler option, you must also use the /NOI linker option. The icc command specifies this linker option by default.

Using the Multithread Library

More than one thread may use the same runtime functions. To avoid contention for internal resources, the library ensures that only one thread at a time is active in the critical section of a function. Although this support is mandatory in a multithread program, it is unnecessary in a single-thread program.

This section describes only the compiler options you use to choose the single-thread or multithread version of the library. There is more information on creating a multithread program in Chapter 11, "Creating Multithread Programs" on page 179.

If you want to create an executable file with multithread capabilities:

1. Specify the /Gm+ option when you compile.
2. Use the multithread library when you link the object files.

If you want to create an executable file designed for a single thread only:

1. Use the default option /Gm- when you compile.
2. Use the single-thread library when you link the object files.

Assuming you used the correct compiler option, the default library for that option is linked. If you override the default libraries with the /NOD linker option, you must explicitly give the name of all libraries you are using on the linker command line.

Enabling Subsystem Development

If you are creating a subsystem, the appropriate libraries are selected when you specify the /Rn option, which is described on page 117.

Functions in the subsystem libraries are intended for use in single-thread applications only. No multithread support is provided. If you want to use the subsystem libraries in multithread programs, you must provide your own protection and serialization using OS/2 semaphores. You must also provide your own buffering for input and output.

See Chapter 17, "Developing Subsystems" on page 303 for information on developing subsystems.

Controlling the Logo Display on Compiler Invocation

By default, the C/C++ Tools logo appears on stderr when the compiler is invoked. You can stop the logo from appearing on icc invocation by specifying the /Q+ option. To request explicitly that the logo appear, specify the /Q- option.

Controlling Stack Allocation and Stack Probes

Under the OS/2 operating system, the stack is fully allocated for the first thread of a process. For all subsequent threads, the operating system allocates the stack as a sparse object that grows in size as required.

Setting the Stack Size

You can set the stack size in one of three ways:

1. Specify the /B"/STACK:*size*" compiler option.
2. Specify the /STACK:*size* parameter on the linker command line.
3. Use a module definition file (.DEF) file for the first thread of an application, and the _beginthread function call for threads created later.

See "Creating a Module Definition File" on page 198 for more information on .DEF files. See the *C Library Reference* for a description of the _beginthread function.

The default stack size is 32K for the first thread. Setting the stack size using one of the options listed above overrides the default value. For example, specifying the linker option

```
/STACK:65536
```

sets the stack size to be 64K.

If your program calls 16-bit code, you can set the stack for the 16-bit code using the #pragma stack16 directive, described in the *Online Language Reference*. Because the 16-bit stack is allocated from the 32-bit stack, you must ensure that the 32-bit stack is large enough for both your 32-bit and 16-bit code.

Automatic Stack Growth

For all threads other than the first, the operating system allocates the stack as a sparse object. The total stack size is the size you specified or the multiple of 4K that is closest to, but greater than, the size you specified. The page with the largest address is committed, and the page below it is set up as a *guard page*. No other pages are committed.

When the guard page is accessed, an *out of stack exception* is generated. The system responds by attempting to get another guard page below the one previously allocated.[1]

If this attempt is successful, the original guard page becomes a normal stack page and the stack grows automatically. This process continues until a new guard page can no longer be allocated.

If the system cannot set a new guard page, because it has reached the size limit passed to the linker by an option or through _beginthread, a *guard page allocation failure* exception is generated. The same exception is generated when the _alloca function runs out of memory.

Note: The last 4K of the stack (the final guard page) is reserved to allow handling of exception conditions. If a guard page exception occurs and not enough stack remains to handle the exception, the program is terminated. For more information about exceptions, see Chapter 18, "Signal and OS/2 Exception Handling" on page 317.

Stack Probes

For the stack growth mechanism to work correctly, each 4K page must be accessed in the correct order. To ensure the correct access, the C/C++ Tools compiler generates one or more stack probes in the prolog of each procedure with automatic storage greater than 2K. (Stack probes start after 2K because exception handling may require up to 2K of stack storage.)

1 For the purposes of this discussion, the stack grows down.

The stack probe instructions allow the guard-page exception mechanism to enlarge the stack if necessary. If an attempt is made to access the stack below the guard page, stack probes ensure that each page of the stack up to that access point is allocated correctly. Without stack probes, accessing the stack below the guard page causes an exception and the process terminates. The compiler ensures that structures greater than 4K that are passed by value are placed on the stack to allow this mechanism to work.

Support for automatic stack growth is provided by default as needed.

Note: The _alloca function allocates storage on the stack. Unless you specify the /Gs+ option, the compiler generates stack probes to allocate the required memory.

You do not need to use stack probes if:

- Your program has only one thread. The stack is fully allocated for the first thread.
- You can guarantee that the stack will always be allocated. For example, you could write a guard routine to run once at the beginning of each thread and serially access each page up to the last page, leaving that page as a guard page.
- Your local variables require less than 2K of storage on the stack.

To turn off stack-probe generation, specify the /Gs+ compiler option. (See “Code Generation Options” on page 111 for the option description.) Because stack probes go into the prolog of every function with more than 2K of stack storage, your program will run faster with the stack probes turned off. If your program only makes occasional uses of large automatic storage and the entire stack has been allocated, not using stack probes may result in inefficient use of available memory.

Chapter 5. Using Compiler Options

You can use compiler options to alter many different aspects of the compilation and linking of your program. This chapter describes the C/C++ Tools compiler options and tells you how to use them.

Specifying Compiler Options

Compiler options are not case sensitive, so you can specify the options in lower-, upper-, or mixed case. For example, you can specify the /Rn option as /rn. You can also substitute a dash (-) for the slash (/) preceding the option. For example, -Rn is equivalent to /Rn. Lower- and uppercase, dashes, and slashes can all be used on one command line, as in:

```
icc /ls -RN -kA /Li prog.c
```

You can specify compiler options in the following ways:

On the command line

Compiler options specified on the command line override any previously specified in the ICC environment variable (as described below and in "OS/2 Environment Variables for Compiling" on page 34).

For example, to compile a source file with the no-optimization option, enter:

```
icc /O- myprog.c
```

If you have more than one source file in your program, see "Compiling Programs with Multiple Source Files" on page 31 for information on specifying options.

In the ICC **environment variable**

Frequently used command-line options can be stored in the ICC environment variable. This method is useful if you find yourself repeating the same command-line options every time you compile. You can also specify source file names in ICC.

Specifying Compiler Options

The ICC environment variable can be set either from the command line, in a command (.CMD) file, or in the CONFIG.SYS file. If it is set on the command line or by running a command file, the options will only be in effect for the current session. If it is set in the CONFIG.SYS file, the options will be in effect every time you use icc unless you override them using a .CMD file or by specifying options on the command line.

For example, to specify that a source listing be generated for all compilations and that the macro DEBUG be defined to be 1, use the following command at the OS/2 prompt (or in your CONFIG.SYS file if you want these options every time you use the compiler):

```
SET ICC=/Ls+ /DDEBUG::1
```

(The double colon must be used because the "=" sign is not allowed in OS/2 environment variables.)

Now, type icc prog1.C to compile prog1.C. The macro DEBUG will be defined as 1, and a source listing will be produced.

Options specified on the command line override the options in the ICC variable. For example, the following compiler invocation voids the effect of the ICC setting in the last example:

```
icc /Ls- /UDEBUG fred.c
```

See "OS/2 Environment Variables for Compiling" on page 34 for more information about using ICC and other environment variables.

In the WorkFrame/2 environment

If you have installed the WorkFrame/2 product, you can set compiler options using the options dialogs. You can use the dialogs when you create or modify a project.

Options you select while creating or changing a project are saved with that project.

For more information on setting options and using the WorkFrame/2 product, refer to the WorkFrame/2 documentation.

Using Parameters with Compiler Options

For all compiler options that take parameters, the following rules apply:

- If a parameter is required, zero or more spaces may appear between the option and the parameter. For example, both /FeMyexe.exe and /Fe Myexe.exe are valid.
- If a parameter is optional, no space is allowed between the option and parameter. For example, /FlMylist.lst is valid, but /Fl Mylist.lst is not.

The syntax of the compiler options varies according to the type of parameter that is used with the option. There are four types of parameters:

- Strings
- File names
- Switches
- Numbers.

Strings

If the option has a string parameter, the string must be enclosed by a pair of double quotation marks if there are spaces in the string. For example, /V"Version 1. " is correct. If there are no spaces in the string, the quotation marks are not necessary. For example, both /VNew and /V"New" are valid.

If the string contains double quotation marks, precede them with the backslash (\) character. For example, if the string is abc"def, specify it on the command line as "abc\"def". This combination is the only escape sequence allowed within string options. Do not end a string with a backslash, as in "abc\".

Do not put a space between the option and the string.

Compiler Option Parameters

File Names

If you want to use a file that is in the current directory, specify only the file name. If the file you want to use is not in the current directory, specify the path and file name. For example, if your current directory is E:\, your source file is E:\myprog.c, and you compile using the defaults, your executable file will be called myprog.exe. If you want to put your executable file into the F:\ directory and call it newprog.exe, use the following command:

```
icc /FeF:\newprog.exe myprog.c
```

If you do not specify an extension for the executable file, .EXE is assumed.

If your file name contains spaces (as permitted by the High Performance File System(HPFS)) or any elements of the HPFS extended character set, the file name must be enclosed in double quotation marks. In such a case, do not put a space between the option and a file name or directory.

Switches

Some options are used with plus (+) or minus (-) signs. If you do not use either sign, the compiler processes the option as if you had used the + sign. When you use an option that uses switches, you can combine them. For example, the following two option specifications have the same result:

```
/La+ /Le+ /Ls+ /Lx-
/Laesx-
```

Note that the - sign applies only to the switch immediately preceding it.

Numbers

When an option uses a number as a parameter, do not put a space between the option and the number. When an option uses two numbers as parameters, separate the numbers with a comma. Do not leave a space between the numbers and the comma. For example:

```
/Sg1 ,132
```

Scope of Compiler Options

The compiler options are categorized according to how they are processed. The categories are:

Local
Global
Cumulative.

Local

A local option applies only to the source files that follow the option. The last, or rightmost, occurrence of these options is the one that is in effect for the source file or files that follow it.

Most compiler options are local. The exceptions are listed under the Global heading. In the following example, the file module1.c will be compiled with the option /Fa- because this option follows /Fa+.

```
icc /Fa+ /Fa- module1.c
```

Note: The /D (define a preprocessor macro) is different from the other local variables in that the **first** definition of a macro is the one that is used. If a preprocessor macro is defined more than once, a warning appears.

Global

A global option applies to all the source files on the command line. If a global option is specified more than once, the last occurrence of the option is the one in effect. A global option can follow the last file on the command line.

The following options are global:

/B /C /Fe /Fm /Gu /H /Mp /Ms /Ol /Q /Sd /Sn

Cumulative

The local option /I and the global option /B have a special characteristic. Each time you specify one of them, the parameters you specify are appended to the parameters previously stated. For example, the command

```
icc /Ia: /Ib:\cde /Ic:\fgh prog.c
```

causes the following search path to be built:

```
a:;b:\cde;c:\fgh
```

ICC Combined with Options Entered on the Command Line

When you specify compiler options both in ICC and on the command line, the compiler evaluates both sets of options. When the compiler is invoked:

1. The string associated with ICC is retrieved.
2. The command line is retrieved.
3. The command line is appended to the ICC string, combining the two into a single command line.
4. This combined command line is read from left to right, and the compiler option precedence rules are applied.
5. The files are compiled and linked using the options as interpreted in the previous step.

Related Options

Some options are required with other options:

- If you specify any of the /Le, /Li, or /Lj options, you must also specify the /Ls option.
- If you specify the /Gr option, you must also specify the /Rn option.

To use EXTRA, you must specify both the /Gh and /Ti options.

Conflicting Options

Some options are incompatible with other options. If options specified on the command line are in conflict, the following rules apply:

- The /Fc option takes precedence over the /Fa, /Fb, /Fe, /Fm, /Fo, /Ft, /Fw, /Ol, /P, /Pc, /Pd, and /Pe options.
- The /P, /Pc, /Pd, and /Pe options take precedence over the /Fa, /Fb, /Fe, /Fl, /Fm, /Fo, and /Ft options, the /Fw, /Gu, and /Ol options, the /Fi and /Si options, and all listing file (/L) options.
- The /Rn option takes precedence over the /Gm, /Sh, and /Sv options.
- The /Fo- option takes precedence over the /Ti option.
- The /C option takes precedence over the /Fe and /Fm options.
- The /O- option takes precedence over the /Os+ options.
- The /Li and /Lj options take precedence over the /Fi and /Si options.
- The /Lj+ option takes precedence over the /Li option.

Language-Dependent Options

Some C/C++ Tools options are only valid when compiling C programs, while others only apply to C++ programs. The following options are valid for C programs only:

/Fd- Store internal work files in shared memory. C++ files must be compiled with /Fd+.

/Gv Control handling of DS and ES registers for virtual device driver development. VDD support is provided for C only.

/K*n* Control diagnostic messages. The /W*grp* options replace the /K*n* options and provide greater control over the messages. The /K*n* options are mapped to the /W*grp* options for you in C programs, but are not supported for C++ programs. The /W*grp* options are supported for both C and C++.

/Sg Set margins for input files. This option is provided primarily for compatibility with IBM C/370. C++ does not require any such compatibility.

Scope of Compiler Options

- /Sq Set sequence numbers for input files. This option is provided primarily for compatibility with IBM C/370. C++ does not require any such compatibility.
- /Sr Set type conversion rules. The C++ language only supports the new type conversion rules defined by the ANSI standard.
- /Ss Allow use of double slashes for comments. C++ allows double slashes to indicate comments as part of the language.
- /S2 Allow only SAA Level 2 C constructs. There is no SAA definition of the C++ language.

The following options are valid for C++ programs only:

- /Fb Control generation of browser files.
- /Ft Control generation of files for template resolution. The C language does not support templates.
- /Gx Control inclusion of C++ exception handling information. The C language does not include specific constructs for exception handling.
- /Sc Allows constructs compatible with earlier versions of the C++ language. These constructs are not allowed in C.

Specifying Options with Multiple Source Files

When you are compiling programs with multiple source files, an option is in effect for all the source files that follow it. For example, if you enter the following command:

```
icc /O+ main.c /Fa sub1.c /Lx /O- sub2.c
```

The file main.c will be compiled with the option /O+.
The file sub1.c will be compiled with the options /O+ and /Fa+.
The file sub2.c will be compiled with the options /O-, /Fa+ and /Lx.

The name of the executable module will be the same as the name of the first source file (main) but with the extension .EXE.

Compiler Options for Presentation Manager Programming

If you are using the C/C++ Tools product to develop PM applications, you will probably want to use the following options:

Option	Description
/Se	Allow all C/C++ Tools language extensions. (This is the default.)
/Gm	Use the multithread libraries.
/Gs-	Do not remove stack probes. (This is the default.)
/Wpro	Produce diagnostic messages about unprototyped functions.

Examples of Compiler Options for Choosing Libraries

Figure 1 on page 80 shows the combinations of compiler options you use to create a particular type of module, according to:

- Static or dynamic linking
- Threading level:
  - Single-thread (/Gm-)
  - Multithread (/Gm+)
- Library being used:
  - Standard (/Re)
  - Subsystem (/Rn)
- Module being built:
  - Executable (/Ge+)
  - DLL (/Ge-)

The defaults used by the compiler are:

- /Gd- (Use static linking)
- /Gm- (Use the single-thread library)
- /Re (Use the standard library)
- /Ge+ (Build an executable module).

Examples of Compiler Options

Figure 1. Combinations or Compiler Options for Specifying Libraries

Linking Type	Threading	Library used	Module Type	Options required in addition to defaults
Static	Single	Standard	EXE	None
Static	Single	Standard	DLL	/Ge-
Static	Multi	Standard	EXE	/Gm+
Static	Multi	Standard	DLL	/Gm+ /Ge-
Static	N/A	Subsystem	EXE	/Rn
Static	N/A	Subsystem	DLL	/Rn /Ge-
Dynamic	Single	Standard	EXE	/Gd+
Dynamic	Single	Standard	DLL	/Gd+ /Ge-
Dynamic	Multi	Standard	EXE	/Gd+ /Gm+
Dynamic	Multi	Standard	DLL	/Gd+ /Gm+ /Ge-
Dynamic	N/A	Subsystem	EXE	/Gd+ /Rn
Dynamic	N/A	Subsystem	DLL	/Gd+ /Rn /Ge-

Compiler Option Classification

The compiler options are divided into groups by function. The following list tells you which options are in each group. For information on each option, see the page numbers listed here.

The tables that follow describe the options grouped by function.

In the tables, the **Default** column states the action the compiler takes if no option is specified; the **Changing Default** column shows how you can change the default.

Where necessary, an option is described in greater detail following the table.

Output File Management Options

Use the options listed in this section to control the files that the compiler produces.

Note: You do not have to specify the plus symbol (+) when specifying an option. For example, the forms /Fa+ and /Fa are equivalent.

Figure 2 (Page 1 of 3). Output File Management Options

Option	Description	Default	Changing Default
/Fa[+|-] /Fa*name*	Produce and name an assembler listing file that has the source code as comments. **Note:** This listing is not guaranteed to compile.	/Fa- Do not create an assembler listing file.	/Fa[+] Create an assembler listing file that has the same name as the source file, with the extension .asm. /Fa*name* Create the listing file *name*.asm.
/Fb[+|-]	Produce a browser file. **Note:** This option is valid for C++ files only.	/Fb- Do not create a browser file.	/Fb[+] Create a browser file. The file has the same name as the source file with the extension .brs.
/Fc[+|-]	Perform syntax check only.	/Fc- Compile and produce output files according to other options.	/Fc[+] Do only a syntax check. The only output files you can produce when this option is in effect are listing (.lst) files.
/Fd[+|-]	Specify work file storage area.	/Fd- Store internal work files in shared memory. **Note:** When you compile C++ code, /Fd+ becomes the default. You cannot specify /Fd- for C++ code.	/Fd[+] Store internal work files on disk in the directory specified by the TMP variable.
/Fe*name*	Specify name of executable file or DLL.	Give the executable file the same name as the first source file, with the extension .exe or .dll.	/Fe*name* Name the executable file *name*.exe or *name*.dll.

Figure 2 (Page 2 of 3). Output File Management Options

Option	Description	Default	Changing Default
/Fi[+|-]	Control creation of precompiled header files.	/Fi- Do not create a precompiled header file.	/Fi[+] Create a precompiled header file if none exists or if the existing one is out-of-date.
/Fl[+|-] /Fl*name*	Produce and name a listing file.	/Fl- Do not create a listing file.	/Fl[+] Give the listing the same file name as the source file, with the extension .lst. /Fl*name* Name the listing file *name*.lst.
/Fm[+|-] /Fm*name*	Produce and name a linker map file.	/Fm- Do not create a map file.	/Fm[+] Create a linker map file with the same file name as the source file, with the extension .map. /Fm*name* Create map file *name*.map. **Note:** Use the /B"/map" option to get a more detailed map file.
/Fo[+|-] /Fo*name*	Produce and name an object file.	/Fo[+] Create an object file, and give it the same name as the source file, with the extension .obj.	/Fo- Do not create an object file. /Fo*name* Create object file *name*.obj.
/Ft[+|-] /Ft*dir*	Control generation of files for template resolution. **Note:** This option is valid for C++ files only. The C language does not support the use of templates.	/Ft+ Generate files for template resolution in the TEMPINC subdirectory under the current directory.	/Ft[-] Do not generate files for template resolution. /Ft*dir* Generate the files and place them in the directory *dir*.

Output File Management Options

Figure 2 (Page 3 of 3). Output File Management Options

Option	Description	Default	Changing Default
/Fw[+|-] /Fw*name*	Control generation and use of intermediate code files.	/Fw- Perform regular compilation; do not save intermediate code files.	/Fw[+] Create intermediate code files only; do not complete compilation. /Fw*name* Create intermediate code files only and name them *name*.w, *name*.wh, and *name*.wi; do not complete compilation.

File Names and Extensions

If you do not specify an extension for the file management options that take a file name as a parameter, the default extension is used. For example, if you specify /Flcome, the listing file will be called come.lst. Although you can specify an extension of your own choosing, you should use the default extensions. See "File Types" on page 33 for more information on default extensions.

If you use an option without using an optional *name* parameter, the name of the following source file and the default extension is used, with the exception of the /Fm option. If you do not specify a name with /Fm, the name of the first file given on the command line is used, with the default extension .map.

Note: If you use the /Fe option, you **must** specify a name or a path for the file. If you specify only a path, the file will have the same name as the first source file on the command line, with the path specified.

See "Using Parameters with Compiler Options" on page 73 for more information on using file names as parameters with options.

Examples of File Management Options

Perform syntax check only:

```
icc /Fc+ myprog.c
```

Name the object file:

```
icc /Fobarney.obj fred.c
```

This names the object file barney.obj instead of the default, fred.obj.

Name the executable file:

```
icc /Febarney.exe fred.c
```

This names the object file barney.exe instead of the default, fred.exe.

Name the listing file:

```
icc /Floutput.my /L fred.c
```

This creates a listing output called output.my instead of fred.lst.

Name the linker map file:

```
icc /Fmoutput.map fred.c
```

This creates a linker map file called output.map instead of fred.map.

Name the assembler listing file:

```
icc /Fabarney fred.c
```

This names the output barney.asm. instead of fred.asm.

#include File Search Options

Use these options to control which paths are searched when the compiler looks for #include files. The paths that are searched are the result of the information in the INCLUDE environment variable and in ICC, combined with how you use the following compiler options.

Figure 3. #include File Search Options

Option	Description	Default	Changing Default
/I*path[;path]*	Specify #include search path(s).	Search directory of source file (for user files only), and then search paths given in the INCLUDE environment variable.	/I*path[;path]* Search *path[;path]*.
/Xc[+|-]	Specify whether to search paths specified using /I.	/Xc- Search paths specified using /I.	/Xc[+] Do not search paths specified using /I.
/Xi[+|-]	Control INCLUDE environment variable search paths.	/Xi- Search the paths specified in the INCLUDE environment variable.	/Xi[+] Do not search the paths specified by the INCLUDE environment variable.

Using the #include File Search Options

The /I option must be followed by one or more directory names. A space may be included between /I and the directory name. If you specify more than one directory, separate the directory names with a semicolon.

If you use the /I option more than once, the directories you specify are appended to the directories you previously specified. For example:

```
/Id:\hdr;e:\    /I  f:\
```

is equivalent to

```
/Id:\hdr\;e:\;f:\;
```

#include File Search Options

If you specify search paths using /I in both the ICC environment variable and on the command line, **all** the paths are searched. The paths specified in ICC are searched before those specified on the command line.

Once you use the /Xc option, the paths previously specified by using /I cannot be recovered. You have to use the /I option again if you want to reuse the paths canceled by /Xc.

The /Xi option has no effect on the /Xc and /I options. For further information on #include files and search paths, see "Controlling #include Search Paths" on page 38.

Listing File Options

The options listed below allow you to control whether or not a listing file is produced, the type of information in the listing, and the appearance of the file.

Note: The following options only modify the appearance of a listing; they do not cause a listing to be produced. Use them with one of the other listing file options or the /Fl option to produce a listing:

/Le /Li /Lj /Lp /Lt /Lu

If you specify any of the /Le, /Li, or /Lj options, you must also specify the /Ls option.

Figure 4 (Page 1 of 2). Listing Output Options

Option	Description	Default	Changing Default
/L[+|-]	Produce a listing file.	/L- Do not produce a listing file.	/L[+] Produce a listing file with only a prolog and error messages.
/La[+|-]	Include a layout of all referenced struct and union variables, with offsets and lengths.	/La- Do not include a layout.	/La[+] Include a layout.
/Lb[+|-]	Include a layout of all struct and union variables.	/Lb- Do not include a layout.	/Lb[+] Include a layout.
/Le[+|-]	Expand all macros.	/Le- Do not expand macros.	/Le[+] Expand macros.
/Lf[+|-]	Set all listing options on or off.	/Lf- Set all listing options off.	/Lf[+] Set all listing options on.
/Li[+|-]	Expand user #include files.	/Li- Do not expand user #include files.	/Li[+] Expand user #include files.
/Lj[+|-]	Expand user and system #include files.	/Lj- Do not expand user and system #include files.	/Lj[+] Expand user and system #include files.

Figure 4 (Page 2 of 2). Listing Output Options

Option	Description	Default	Changing Default
/Lp*num*	Set page length.	/Lp66 Set page length to 66 lines.	/Lp*num* Specify *num* lines per page of listing. *num* must be between 15 and 65535 inclusive.
/Ls[+|-]	Include the source code.	/Ls- Do not include the source code.	/Ls[+] Include the source code.
/Lt"*string*"	Set title string.	Set title string to the name of the source file.	/Lt"*string*" Set title string to *string*. Maximum string length is 256 characters.
/Lu"*string*"	Set subtitle string.	/Lu"" Set no subtitle (null string).	/Lu"*string*" Set subtitle string to *string*. Maximum string length is 256 characters.
/Lx[+|-]	Generate a cross-reference table of referenced variable, structure, and function names, that shows line numbers where names are declared.	/Lx- Do not generate a cross-reference table.	/Lx[+] Generate a cross-reference table.
/Ly[+|-]	Generate a cross-reference table of all variable, structure, and function names, plus all local variables referenced by the user.	/Ly- Do not generate a cross-reference table.	/Ly[+] Generate a cross-reference table.

Note: You can also specify titles using the #pragma title and subtitle directives, but these titles do not appear on the first page of the listing output.

Including Information about Your Source Program

You can use three options to include information about your source program in the listing file:

/Ls[+] Includes your source program in the listing file.

/Li[+] Shows the included text after the user #include directives.

/Lj[+] Shows the included text after both user and system #include directives.

If you use HPFS and have very long file names, there may not be enough room for the file names on the lines showing the included code. Counters are used in the INCLUDE column of the listing output, and the file name corresponding to each number is given at the bottom of the source listing.

Note: The /Li and /Lj options do not work in combination. If you specify the /Lj option, /Li[+] and /Li- have no effect.

Including Information about Variables

The options that produce information about the variables used in your program provide the following amount of detail:

/La[+] Includes a table of all the referenced struct and union variables in the source program. The table shows how each structure and union in the program is mapped. It contains the following information:

- The name of the structure or union and the elements within each.
- The byte offset of each element from the beginning of the structure or union. The bit offset for unaligned bit data is also given.
- The length of each element.
- The total length of each structure, union, and substructure in both packed and unpacked formats.

| /Lb[+] Includes a table of all struct and union variables in the program. The table contains the same type of information as the one generated by /La[+].

/Le[+] Includes all expanded macros in the listing file.

/Lx[+] Includes a cross-reference table that contains a list of the referenced identifiers in the source file together with the numbers of the lines in which they appear.

/Ly[+] Includes a cross-reference table that contains a list of all identifiers referenced by the user and all external identifiers, together with the numbers of the lines in which they appear.

Debugging and Diagnostic Information Options

The options listed here are useful for debugging your programs.

Note: The /W*grp* options and /K*n* options generate the same messages. The /W*grp* options give you greater control over the types of messages generated. The /K*n* options are provided for compatibility with C Set/2 V1.0 only, and are mapped for you to the correct /W*grp* options. They are not supported for use in C++ programs, and will not be supported in future versions of the C/C++ Tools product.

The information generated by the C/C++ Tools debugger and the /K*n* and /W*grp* options is provided to help you diagnose problems in your code. Do not use the diagnostic information as a programming interface.

Figure 5 (Page 1 of 4). Debugging Options

Option	Description	Default	Changing Default
/Ka[+|-]	Control messages about variable assignments that can result in a loss of precision. Maps to the /Wtrd option.	/Ka- Suppress messages about assignments that may cause a loss of precision.	/Ka[+] Produce messages about inappropriate assignments of long values.
/Kb[+|-]	Control messages about basic diagnostics generated by /K options. Maps to the /Wgen option. **Note:** Many of the messages considered general diagnostics in the C Set/2 V1.0 product are now controlled by a specific /W option.	/Kb- Suppress basic diagnostic messages.	/Kb[+] Produce basic diagnostic messages.

Figure 5 (Page 2 of 4). Debugging Options

Option	Description	Default	Changing Default
/Kc[+|-]	Control preprocessor warning messages. Maps to the /Wppc option.	/Kc- Suppress preprocessor warning messages.	/Kc[+] Produce preprocessor warning messages.
/Ke[+|-]	Control messages about enum usage. Maps to the /Wenu option.	/Ke- Suppress messages about enum usage.	/Ke[+] Produce messages about enum usage.
/Kf[+|-]	Set all diagnostic messages options on or off. Maps to the /Wall option.	/Kf- Set all diagnostic messages options off.	/Kf[+] Set all diagnostic messages options on.
/Kg[+|-]	Control messages about the appearance and usage of goto statements. Maps to the /Wgot option.	/Kg- Suppress messages about goto statements.	/Kg[+] Produce messages about goto statements.
/Ki[+|-]	Control messages about variables that are not explicitly initialized. Maps to the /Wini and /Wuni options.	/Ki- Suppress messages about uninitialized variables.	/Ki[+] Produce messages about uninitialized variables.
/Ko[+|-]	Control diagnostic messages about portability. Maps to the /Wpor option.	/Ko- Suppress portability messages.	/Ko[+] Produce portability messages.
/Kp[+|-]	Control messages about function parameters that are not used. Maps to the /Wpar option.	/Kp- Suppress messages about unused function parameters.	/Kp[+] Produce messages about unused function parameters.

Debugging and Diagnostic Options

Figure 5 (Page 3 of 4). Debugging Options

Option	Description	Default	Changing Default
/Kr[+|-]	Control messages about mapping of names to the linkage editor. Maps to the /Wtru option.	/Kr- Suppress messages about name mapping.	/Kr[+] Produce messages about name mapping.
/Kt[+|-]	Control preprocessor trace messages. Maps to the /Wppt option.	/Kt- Suppress preprocessor trace messages.	/Kt[+] Produce preprocessor trace messages.
/Kx[+|-]	Control messages about variables and functions that have external declarations, but are never used. Maps to the /Wext and /Wuse options.	/Kx- Suppress messages about unreferenced external variables and functions.	/Kx[+] Produce messages about unreferenced external variables and functions.
/N*n*	Set maximum number of errors before compilation aborts.	Set no limit on number of errors.	/N*n* End compilation when error count reaches *n*.
/Ti[+|-]	Generate C/C++ Tools debugger information.	/Ti- Do not generate debugger information.	/Ti[+] Generate debugger information.
/Ts[+|-]	Generate code to allow the debugger to maintain the call stack across all calls that do not chain the EBP, that is, system calls.	/Ts- Do not generate code to allow the debugger to maintain the call stack.	/Ts[+] Generate code to allow the debugger to maintain the call stack.
/Tx[+|-]	Control information generated when an exception occurs.	/Tx- Provide only the exception message and address when an exception occurs; do not provide a complete machine-state dump.	/Tx[+|-] Provide a complete machine-state dump when an exception occurs.

Figure 5 (Page 4 of 4). Debugging Options

Option	Description	Default	Changing Default
/W<*grp*>[+|-][*grp*]	Control diagnostic messages.	/Wall- Do not generate diagnostic messages.	/W*grp* Generate messages in the *grp* group. More than one group may be specified. See Using the /W*grp* Diagnostic Options which follows for descriptions of the different groups of messages.
/W[|1|2|3]	Set the type of message the compiler produces and that causes the error count to increment.	/W3 Produce all message types.	/W Produce only severe errors. /W1 Produce severe errors and errors. /W2 Produce severe errors, errors, and warnings.

Note: If you use the /Ti option to generate debugger information, it is recommended that you turn optimization off (/O-). (This recommendation does not apply if you are using /Ti to generate information for EXTRA.) Because of the nature of the optimizations performed, many of the functions of the debugger will not operate properly on optimized code. Because the compiler produces debugging information as if the code were not optimized, the information may not accurately describe an optimized program being debugged.

Because of the effects of optimization, debugging information generated with optimization is limited to setting breakpoints at function entry and function exit and stepping through the program at assembly level. Accurate symbol and type information is not always available.

To make full use of the C/C++ Tools debugger, set optimization off and use the /G3 option. (Note that these are the defaults.)

Using the /W*grp* Diagnostic Options

Use these options to examine your source code for possible programming errors, weak programming style, and other information about the structure of your program. When you specify /Wall[+], all suboptions are turned on and all possible diagnostic messages are reported. Because even a simple program that contains no errors can produce many informational messages, you may not want to use /Wall very often. You can use the suboptions alone or in combination to specify the type of messages that you want the compiler to report. Suboptions can be separated by an optional + sign. To turn off a suboption, you must place a - sign after it. You can also combine the /W[|1|2|3] options with the /W*grp* options.

The following table lists the message groups and the message numbers that each controls, as well as the /K*n* option that formerly controlled each message. Messages generated for C files begin with EDC0, while messages for C++ files begin with EDC3.

Figure 6 (Page 1 of 3). /Wgrp Options

grp	/K*n* **Option**	**Controls Messages About**	**Messages**
all	/Kf	All diagnostics.	All message numbers listed in this table.
cls	(none)	Use of classes	EDC3110, EDC3253, EDC3266
cmp	(none)	Possible redundancies in unsigned comparisons.	EDC3138, EDC3139, EDC3140
cnd	/Kb	Possible redundancies or problems in conditional expressions.	EDC0816, EDC0821, EDC0822, EDC3107, EDC3130
cns	/Kb	Operations involving constants.	EDC0823, EDC0824, EDC0838, EDC0839, EDC0865, EDC0866, EDC0867, EDC3131, EDC3219
cnv	/Kb	Conversions.	EDC3313
cpy	(none)	Problems generating copy constructors.	EDC3199, EDC3200
eff	/Kb	Statements with no effect.	EDC0811, EDC0812, EDC0813, EDC0814, EDC0815, EDC3165, EDC3215

Figure 6 (Page 2 of 3). /Wgrp Options

grp	/K*n* Option	Controls Messages About	Messages
enu	/Ke	Consistency of enum variables.	EDC0830, EDC0831, EDC3137
ext	/Kb and /Kx	Unused external definitions.	EDC0803, EDC0804, EDC0810, EDC3127
gen	/Kb	General diagnostics.	EDC0807, EDC0809, EDC0826, EDC0835, EDC0868, EDC0869, EDC3101
gnr	(none)	Generation of temporary variables.	EDC3151
got	/Kg	Usage of goto statements.	EDC0832, EDC0837
ini	/Ki	Possible problems with initialization.	EDC0861, EDC0862, EDC0863, EDC0864
lan	(none)	Effects of the language level.	EDC3116
obs	/Kb	Features that are obsolete.	EDC0827, EDC0828
ord	/Kb	Unspecified order of evaluation.	EDC0829
par	/Kp	Unused parameters.	EDC0800, EDC3126
por	/Ko, /Kb	Nonportable language constructs.	EDC0464, EDC0819, EDC0820, EDC3132, EDC3133, EDC3135, EDC3136, EDC3307
ppc	/Kc	Possible problems with using the preprocessor.	EDC0836, EDC0841, EDC0842, EDC0843, EDC0844, EDC0845, EDC0846, EDC0847, EDC0848
ppt	/Kt	Trace of preprocessor actions.	EDC0851, EDC0852, EDC0853, EDC0854, EDC0855, EDC0856, EDC0857, EDC0858, EDC0859, EDC0860, EDC0870
pro	/Kb	Missing function prototypes.	EDC0185
rea	/Kb	Code that cannot be reached.	EDC0825, EDC3119
ret	/Kb	Consistency of return statements.	EDC0833, EDC0834, EDC3128
trd	/Ka	Possible truncation or loss of data or precision.	EDC0817, EDC0818, EDC3108, EDC3135, EDC3136
tru	/Kr	Variable names truncated by the compiler.	EDC0244

Figure 6 (Page 3 of 3). /Wgrp Options

grp	/K*n* **Option**	**Controls Messages About**	**Messages**
und	(none)	Casting of pointers to or from an undefined class.	EDC3098
uni	/Ki	Uninitialized variables.	EDC0808
use	/Kb, /Kx	Unused auto and static variables.	EDC0801, EDC0802, EDC0805, EDC0806, EDC3002, EDC3099, EDC3100
vft	(none)	Generation of virtual function tables.	EDC3280, EDC3281, EDC3282

More information about the messages generated by the /W*grp* options is available in the *Online Language Reference*.

Examples of /W*grp* Options

Produce all diagnostic messages:

```
icc /Wall blue.c
icc /Wall+ blue.c
```

Produce diagnostic messages about:

- Consistency of declarations
- Unreferenced parameters
- Missing function prototypes
- Uninitialized variables

by turning on the appropriate suboptions:

```
icc /Wdcl+par+pro+uni blue.c
icc /Wdclparprouni blue.c
```

Produce all diagnostic messages except:

- Warnings about assignments that can cause a loss of precision
- Preprocessor trace messages
- External variable warnings

by turning **on** all options, and then turning **off** the ones you do not want:

```
icc /Wall+trd-ppt-ext- blue.c
```

Produce only basic diagnostics, with all other suboptions turned off:

```
icc /Wgen+ blue.c
```

Produce only basic diagnostics and suppress all messages with a severity of "informational" (/W2):

```
icc /Wgen2 blue.c
```

Source Code Options

These options allow you to control how the C/C++ Tools compiler interprets your source file. This control is especially useful, for example, if you are concerned with migrating code or ensuring consistency with a particular language standard.

Figure 7 (Page 1 of 5). Source Code Options

Option	Description	Default	Changing Default
/S[a|c|e|2]	Set language level. See "Setting the Source Code Language Level" on page 41.	/Se Allow all C/C++ Tools language extensions.	/Sa Conform to ANSI standards. /Sc Allow constructs compatible with older levels of the C++ language. See "Setting the Source Code Language Level" on page 41 for details on the constructs allowed. **Note:** This option is valid only for C++ files. /S2 Conform to SAA Level 2 standards. **Note:** This option is valid only for C files.
/Sd[+|-]	Set default file extension. See "Using the /Sd Option" on page 105 for more information.	/Sd- Set the default file extension as .obj.	/Sd[+] Set the default file extension as .c.

Figure 7 (Page 2 of 5). Source Code Options

<table>
<tr><th>Option</th><th>Description</th><th>Default</th><th>Changing Default</th></tr>
<tr><td>/Sg[<i>l</i>][,<<i>r</i>| >]

/Sg-</td><td>Set left and right margins of the input file and ignore text outside these margins. Useful when using source files created on other systems that contain characters that you want to ignore.

Note: This option is only valid for C files.</td><td>/Sg-
Do not set any margins: use the entire input file.</td><td>/Sg[<i>l</i>][,<i>r</i>|]
Set left margin to <i>l</i>. The right margin can be the value <i>r</i>, or an asterisk can be used to denote no right margin. <i>l</i> and <i>r</i> must be between 1 and 65535 inclusive, and <i>r</i> must be greater than or equal to <i>l</i>.</td></tr>
<tr><td>/Sh[+|-]</td><td>Allow use of ddnames.</td><td>/Sh-
Do not allow ddnames.</td><td>/Sh[+]
Allow use of ddnames.</td></tr>
<tr><td>/Si[+|-]</td><td>Control use of precompiled header files.</td><td>/Si-
Do not use precompiled header files.</td><td>/Si[+]
Use precompiled header files if they exist and are current.</td></tr>
<tr><td>/Sm[+|-]</td><td>Control compiler interpretation of unsupported 16-bit keywords, such as near and far.</td><td>/Sm-
Treat unsupported 16-bit keywords like any other identifier.</td><td>/Sm[+]
Ignore unsupported 16-bit keywords.</td></tr>
<tr><td>/Sn[+|-]</td><td>Allow use of double-byte character set (DBCS).</td><td>/Sn-
Do not allow DBCS.</td><td>/Sn[+]
Allow use of DBCS.</td></tr>
<tr><td>/Sp[1|2|4]</td><td>Specify alignment or packing of data items within structures and unions.</td><td>/Sp4
Align structures and unions along 4-byte boundaries (normal alignment).</td><td>/Sp[1|2]
Align structures and unions along 1-byte or 2-byte boundaries. /Sp is equivalent to /Sp1.</td></tr>
</table>

Source Code Options

Figure 7 (Page 3 of 5). Source Code Options

Option	Description	Default	Changing Default
/Sq[*l*][,<*r* | >] /Sq-	Specify columns in which sequence numbers appear, and ignore text in those columns. This option can be used when importing source files from systems that use sequence numbers. **Note:** This option is only valid for C files.	/Sq- Use no sequence numbers.	/Sq[*l*][,*r*] Sequence numbers appear between columns *l* and *r* of each line in the input source code. *l* and *r* must be between 1 and 65535 inclusive, and *r* must be greater than or equal to *l*. If you do not want to specify a right column, use an asterisk for *r*.
/Sr[+|-]	Set type conversion rules. **Note:** This option is valid for C files only.	/Sr- Use new-style rules for type conversion. New-style rules preserve accuracy.	/Sr[+] Use old-style rules for type conversion. Old-style rules preserve the sign. They do not conform to ANSI standards.
/Ss[+|-]	Allow use of double slashes (//) for comments. **Note:** This option is only valid for C files. C++ allows double slashes to indicate comments as part of the language.	/Ss- Do not allow double slashes to indicate comments.	/Ss[+] Allow the double slash format to indicate comments. This type of comment is ended by a carriage return.

Figure 7 (Page 4 of 5). Source Code Options

Option	Description	Default	Changing Default
/Su[+|-|1|2|4]	Control size of enum variables.	/Su- Use the SAA rules, that is, make all enum variables the size of the smallest integral type that can contain all variables.	/Su[+] Make all enum variables 4 bytes. /Su1 Make all enum variables 1 byte. /Su2 Make all enum variables 2 bytes. /Su4 Make all enum variables 4 bytes.
/Sv[+|-]	Allow use of memory files.	/Sv- Do not allow memory files.	/Sv[+] Allow use of memory files.
/Tc	Specify that the following file is a C file. **Important:** The /Tc option **must** be immediately followed by a file name, and applies only to that file.	Compile .cpp and .cxx files as C++ files, and .c and all other unrecognized files as C files.	/Tc Compile the following file as a C source file, regardless of its extension.

Figure 7 (Page 5 of 5). Source Code Options

Option	Description	Default	Changing Default
/Td[c|p]	Specify default language (C or C++) for files.	/Td Compile .cpp and .cxx files as C++ files, and .c and all other unrecognized files as C files.	/Tdc Compile all source and unrecognized files that follow on the command line as C files. /Tdp Compile all source and unrecognized files that follow on the command line as C++ files, and ensure that template functions are resolved correctly. See "Using the /Tdp Option for Template Resolution" on page 106 below. **Note:** You can specify /Td anywhere on the command line to return to the default rules for the files that follow it.
/Tp	Specify that the following file is a C++ file. **Important:** The /Tp option **must** be immediately followed by a file name, and applies only to that file.	Compile .cpp and .cxx files as C++ files, and .c and all other unrecognized files as C files.	/Tp Compile the following file as a C++ source file, regardless of its extension.

Using the /Sd Option

This option specifies whether a file without an extension should be considered a C source file or an object file, and whether it should be compiled and linked or just linked. When using the default (/Sd-), you must specify the extension when using a source file:

```
icc anthony.c
icc efrem.cpp
```

If you omit the extension, the C/C++ Tools compiler assumes that the file is an object file (.obj) and does not compile it, but only invokes the linker. The following commands are equivalent (assuming that /Sd+ has not been specified elsewhere, such as in ICC).

```
icc dale
icc dale.obj
icc /Sd- dale
```

If you want the default file extension to be the default source file extension, use the /Sd+ option. For example, the following two commands are equivalent:

```
icc alistair.c
icc /Sd+ alistair
```

Note: The /Tc and /Tp options override the setting of /Sd. If you specify either /Tc or /Tp followed by a file name without an extension, the compiler looks for the name specified, **without an extension**, and treats the file as a C file (if /Tc was specified) or a C++ file (if /Tp was specified). For example, given the following command:

```
icc /Tp xiaohu
```

the compiler searches for the file xiaohu and compiles it as a C++ file.

Using the /Tdp Option for Template Resolution

When you link C++ object or intermediate code files, you must use icc to invoke the linker and you must specify the /Tdp option. For example:

```
icc /Tdp /Ol tammy.w trish.w
icc /Tdp wang.obj
```

This ensures that any template functions you use are resolved correctly, among other things. You can use the /B option to pass options to the linker.

Preprocessor Options

The options listed here let you control the use of the preprocessor.

Note that the /Pc, /Pd, and /Pe options are actually suboptions of /P. Specifying /Pc- is the same as specifying /P+c- and causes the preprocessor only to be run.

Figure 8 (Page 1 of 3). Preprocessor Options

Option	Description	Default	Changing Default
/D*name*[::*n*] /D*name*[=*n*]	Define preprocessor macros to specified values.	Define no macros on command line.	/D*name*[::*n*] or /D*name*[=*n*] Define preprocessor macro *name* to the value *n*. If *n* is omitted, the macro is set to a null string. Macros defined on the command line override macros defined in the source code.
/P[+|-]	Control the preprocessor.	/P- Run the preprocessor and compiler. Do not generate preprocessor output.	/P[+] Run the preprocessor only. Create a preprocessor output file that has the same name as the source file, with the extension .i.

Preprocessor Options

Figure 8 (Page 2 of 3). Preprocessor Options

Option	Description	Default	Changing Default
/Pc[+|-]	Preserve source code comments in preprocessor output.	/P- Run the preprocessor and compiler. Do not generate preprocessor output.	/Pc- Run the preprocessor only. Create a preprocessor output file and strip out any comments. The output file has the same name as the source file with the extension .i. /Pc[+] Run the preprocessor only. Create a preprocessor output file that includes the comments from the source code. The output file has the same name as the source file with the extension .i.
/Pd[+|-]	Redirect preprocessor output.	/P- Run the preprocessor and compiler. Do not generate preprocessor output.	/Pd- Run the preprocessor only. Do not redirect preprocessor output. Write preprocessor output to a file that has the same name as the source file, with the extension .I. /Pd[+] Run the preprocessor only. Send the preprocessor output to stdout.

Figure 8 (Page 3 of 3). Preprocessor Options

Option	Description	Default	Changing Default
/Pe[+|-]	Suppress #line directives in preprocessor output.	/P- Run the preprocessor and compiler. Do not generate preprocessor output.	/Pe- Run the preprocessor only. Generate #line directives in the preprocessor output. The output file has the same name as the source file with the extension .i. /Pe[+] Run the preprocessor only. Suppress creation of #line directives in preprocessor output. The output file has the same name as the source file with the extension .i.
/U<*name*| >	Undefine macros.	Retain macros.	/U*name* Undefine macro *name*. /U Undefine all macros. **Note:** /U does not affect the macros __DATE__, __TIME__, __TIMESTAMP__, __FILE__, and __FUNCTION, nor does it undefine macros defined in source code.

Using the Preprocessor

Preprocessor directives, such as #include, allow you to include C or C++ code from another source file into yours, to define macros, and to expand macros. See the *C Language Reference* for a list of preprocessor directives and information on how to use them.

If you run only the preprocessor, you can use the preprocessor output (which has all the preprocessor directives executed, but no code compiled) to debug your program. For example, all macros are expanded, and the code for all files included by #include directives appears in your program.

By default, comments in the source code are not included in the preprocessor output. To preserve the comments, use the /Pc option. For C programs, if you use // to begin your comments, you must also specify the /Ss option to include those comments in the preprocessor output.

The /P, /Pc, /Pd, and /Pe options can be used in combination with each other. For example, to preserve comments, suppress #line directives, and redirect the preprocessor output to stdout, specify /Pcde.

Code Generation Options

These options allow you to specify the type of code that the compiler will produce. The types of code include:

- Dynamically linked runtime libraries (See Chapter 12, "Building Dynamic Link Libraries" on page 195.)
- Statically linked runtime libraries
- Single-thread programs
- Multithread programs (See Chapter 11, "Creating Multithread Programs" on page 179.)
- Subsystems. (See Chapter 17, "Developing Subsystems" on page 303.)

Notes:

1. The /Oi[+] option is more effective when /O[+] is also specified.
2. Using optimization (/O[+]) limits your use of the C/C++ Tools debugger to debug your code. The /Ti option is not recommended for use with optimization.

Figure 9 (Page 1 of 7). Code Generation Options

Option	Description	Default	Changing Default
/Gd[+|-]	Specify static or dynamic linking of the runtime library.	/Gd- Statically link the runtime library. All external names beginning with the letters Dos, Kbd, and Vio are reserved. This restriction does not apply when compiling with /Gd+.	/Gd[+] Dynamically link to the runtime library.
/Ge[+|-]	Specify creation of an .EXE or a .DLL file.	/Ge[+] Build an .EXE file.	/Ge- Build a .DLL file.

Figure 9 (Page 2 of 7). Code Generation Options

Option	Description	Default	Changing Default
/Gf[+|-]	Specify fast floating-point execution. If your program does not need to abide by ANSI rules regarding the processing of double and float types, you can use this option to increase your program's performance. Because the fast floating-point method does not perform all the conversions specified by the ANSI standards, the results obtained may differ from results obtained using ANSI methods, but are often more precise.	/Gf- Do not use fast floating-point execution.	/Gf[+] Use fast floating-point execution.
/Gh[+|-]	Generate code enabled for EXTRA and other profiling tools.	/Gh- Do not enable code for EXTRA.	/Gh[+] Enable code to be run by EXTRA and other profiling tools by generated profiler hooks in function prologs. **Note:** To enable code for EXTRA, you must also specify /Ti.

Figure 9 (Page 3 of 7). Code Generation Options

Option	Description	Default	Changing Default
/Gi[+|-]	Specify fast integer execution. If you are shifting bits by a variable amount, you can use fast integer execution to ensure that for values greater than 31, the bits are shifted by the result of a modulo 32 of the value. Otherwise, the result of the shift is 0. **Note:** If your shift value is a constant greater than 32, the result will always be 0.	/Gi- Do not use fast integer execution.	/Gi[+] Use fast integer execution.
/Gm[+|-]	Choose single or multithread libraries.	/Gm- Link with the single-thread version of the library (no multithread capabilities).	/Gm[+] Link with the multithread version of the library.
/Gn[+|-]	Control generation of default library information in object.	/Gn- Provide linker information about the default libraries according to other /G options.	/Gn[+] Do not provide linker information about default libraries. All libraries must be explicitly specified at link time.
/Gr[+|-]	Generate object code that runs at ring 0. Use this option if you are writing code, such as device drivers or operating systems, that will run at ring 0 instead of ring 3.	/Gr- Do not allow object code to run at ring 0.	/Gr[+] Allow object code to run at ring 0.
/Gs[+|-]	Remove stack probes from the generated code.	/Gs- Do not remove stack probes.	/Gs[+] Remove stack probes.

Code Generation Options

Figure 9 (Page 4 of 7). Code Generation Options

Option	Description	Default	Changing Default
/Gt[+|-]	Enable tiled memory and store variables such that they may be passed to 16-bit functions.	/Gt- Do not enable variables to be passed to 16-bit functions.	/Gt[+] Enable all variables to be passed to 16-bit functions. Static and external variables are mapped into 16-bit segments. Variables larger than 64K will be aligned on, but will still cross, 64K boundaries. When this option is specified, the memory management functions calloc, free, malloc, and realloc are mapped to the tiled versions _tcalloc, _tfree, _tmalloc, and _trealloc.
/Gu[+|-]	Tell intermediate linker whether external functions use data defined in the intermediate link.	/Gu- External functions may use data defined in the intermediate files being linked.	/Gu[+] The data is used only within the intermediate files being linked, with the exception of data that is exported using _Export, #pragma export, or a .DEF file. See "Using the Intermediate Code Linker" on page 52 for more information about the intermediate code linker.
/Gv[+|-]	Control handling of DS and ES registers for virtual device driver development. **Note:** This option is valid for C files only. Virtual device driver development is not supported for C++ programs.	/Gv- Do not perform any special handling of the DS and ES registers.	/Gv[+] Save the DS and ES registers on entry to an external function, set them to the selector for DGROUP, then restore them on exit from the function. For more information on developing virtual device drivers, see Chapter 15, "Developing Virtual Device Drivers" on page 281

Figure 9 (Page 5 of 7). Code Generation Options

Option	Description	Default	Changing Default
/Gw[+|-]	Control generation of FWAIT instruction after each floating-point load instruction.	/Gw- Do not generate FWAIT instruction after each floating-point load instruction.	/Gw[+] Generate FWAIT instruction after each floating-point load instruction. This allows the program to take a floating-point stack overflow exception immediately after the load instruction that caused it. **Note:** This option is not recommended because it increases the size of your executable file and greatly decreases its performance.
/Gx[+|-]	Controls removal of C++ exception handling information. **Note:** This option is valid for C++ files only.	/Gx- Do not remove C++ exception handling information.	/Gx[+] Remove C++ exception handling information.
/G[3|4|5]	Specify type of processor.	/G3 Optimize code for use with a 386 processor. The code will run on a 486 or Pentium microprocessor. The compiler includes any 486 or Pentium microprocessor optimizations that do not detract from the performance on the 386 processor. If you do not know what processor your application will be run on, use this option.	/G4 Optimize code for use with a 486 processor. The code will run on a 386 or Pentium microprocessor. The compiler includes any Pentium microprocessor optimizations that do not detract from the performance on the 486 processor. /G5 Optimize code for use with a Pentium Microprocessor. The code will run on a 386 or 486 processor.

Code Generation Options

Figure 9 (Page 6 of 7). Code Generation Options

Option	Description	Default	Changing Default
/M[p|s]	Set calling convention. See Chapter 14, "Calling Conventions" on page 237 for more information.	/Mp Use _Optlink linkage for functions. You must include the Toolkit header files to call OS/2 APIs.	/Ms Use _System linkage for functions. You must include the C/C++ Tools library header files to call C/C++ Tools functions.
/Nd*name*	Specify names of default data and constant segments.	Use the default names DATA32 and CONST32.	/Nd*name* Use the names *name*DATA32 and *name*CONST32. You can then give the segments special attributes. The renamed segments are not placed in the default data group.
/Nt*name*	Specify name of default code or text segment.	Use the default name CODE32.	/Nt*name* Use the name *name*CODE32. You can then give the segment special attributes.
/O[+|-]	Control optimization.	/O- Do not optimize code.	/O[+] Optimize code.
/Oi[+|-|] /Oi*value*	Control inlining of user code.	/Oi- Do not inline any user code. **Note:** When /O+ is specified, /Oi+ becomes the default.	/Oi[+] Inline all user functions qualified with the _Inline or inline keyword. /Oi*value* Inline all user functions qualified with the _Inline or inline keyword or that are smaller than *value* in abstract code units. See "Inlining User Code" on page 56 for more information.

Figure 9 (Page 7 of 7). Code Generation Options

Option	Description	Default	Changing Default
/Ol[+|-]	Control use of intermediate code linker.	/Ol- Do not pass code through the intermediate linker.	/Ol[+] Pass code through the intermediate linker before generating an object file. See "Using the Intermediate Code Linker" on page 52 for more information.
/Om[+|-]	Control size of working set for compiler. See the READ.ME file for a complete description of this option.	/Om- Do not limit working set size.	/Om[+] Limit working set size to approximately 35M.
/Op[+|-]	Control disabling of optimizations involving the stack pointer.	/Op+ Perform optimizations involving the stack pointer.	/Op- Do not perform optimizations that involve the stack pointer. Code that directly manipulates the stack pointer should be compiled with this option. This option is not recommended because it decreases the performance of your executable file.
/Os[+|-]	Control use of instruction scheduler.	/Os- Do not invoke the instruction scheduler. **Note:** When /O+ is specified, /Os+ becomes the default.	/Os+ Invoke the instruction scheduler. **Note:** You cannot specify /Os+ and /O-.
/R[e|n]	Control executable runtime environment.	/Re Generate executable code that runs in a C/C++ Tools runtime environment.	/Rn Generate executable code that can be used as a subsystem without a runtime environment.

Using the /Ge Option

The C/C++ Tools libraries provide two initialization routines, one for executable modules and one for DLLs. For each object file, the compiler must include a reference to the appropriate initialization routine. The name of this routine is then passed to the linker when the file is linked. Use the /Ge option at compile time to tell the compiler which routine to reference.

The /Ge- option causes the compiler to generate a reference to _dllentry for every module compiled. The /Ge+ option generates a reference to _exeentry only if a main function is found in the source. If no main function is included, no linking reference is generated.

If you want to create a library of objects that can be linked into either an executable file or a DLL, use the /Ge+ option when you compile. Typically, none of these objects would contain a reference to main.

If one of the objects **did** contain a reference to main, you can override the /Ge option when you link your files. Create a source file that defines the routine already referenced in your object file. In the same file, add a dummy statement that references the correct initialization routine. Then compile this file and link it with your other object files.

For example, if you compiled tammy.obj using the /Ge+ option, but want to link it to create a DLL, your extra source file would contain statements like the following:

```
int _exeentry = 1;
extern int _dllentry;

int main(void)
{
    int  x;

   ⋮
     x = _dllentry;

   ⋮
}
```

The reference to _exeentry in tammy.obj is resolved by this file, and this file's reference to _dllentry causes the linker to link in the correct initialization routine.

Other Options

Use these options to control linker parameters, logo display, default char type, and other C/C++ Tools options.

Figure 10. Other Options

Option	Description	Default	Changing Default
/B"*options*"	Specify parameters to be passed to linker. See the *Toolkit Tools Reference* for information about the options you can pass to the LINK386 linker.	/B"" Pass only the icc default parameters to the linker. See "Linking Independently of the Compiler" on page 123 for a description of the options passed to the linker by default.	/B"*options*" Pass *options* string to the linker as parameters. The icc default parameters are also passed.
/C[+|-]	Perform compile only, or perform compile and link.	/C- Perform compile and invoke linker.	/C[+] Perform compile only, no link.
/H*num*	Set significant length of external names.	/H255 Set the first 255 characters of external names to be significant.	/H*num* Set the first *num* characters of external names to be significant. The value of *num* must be between 6 and 255 inclusive.
/J[+|-]	Set default char type.	/J[+] Set unspecified char variables to unsigned char.	/J- Set unspecified char variables to signed char.
/Q[+|-]	Display compiler logo when invoking compiler.	/Q- Display logo on stderr.	/Q[+] Do not display logo.
/V"*string*"	Include a version string in the object and executable files.	/V"" Set no version string.	/V"*string*" Set version string to *string*. The length of the string can be up to 256 characters.
?	Display list of compiler options with descriptions.	Compile and produce output files according to other options.	? Display list of compiler options with descriptions.

Examples of Other Options

Passing a parameter to the linker:

```
icc /B"/NOI" fred.c
```

The /NOI option tells the linker to preserve the case of external names in fred.obj.

Imbedding a version string or copyright:

```
icc /V"Version 1. " fred.c
```

This imbeds the version notice in fred.obj.

Other Options

Chapter 6. Finishing Your Program

This chapter describes the linker and other tools and how to invoke them.

Once the compiler has created object modules out of your source files, use the linker to link them together with the C/C++ Tools runtime libraries to create an executable module or DLL. By default, icc invokes the linker for you. To compile and link in separate steps, use the /C+ option to force icc to perform only the compile step. You can then invoke LINK386 separately to link your program.

If you are creating an application for others to use, you must consider that the machines your program will run on may not have access to the same resources your machine has. As a result, you may also need to invoke the resource compiler, message binding, and help facilities for your program.

Linking Independently of the Compiler

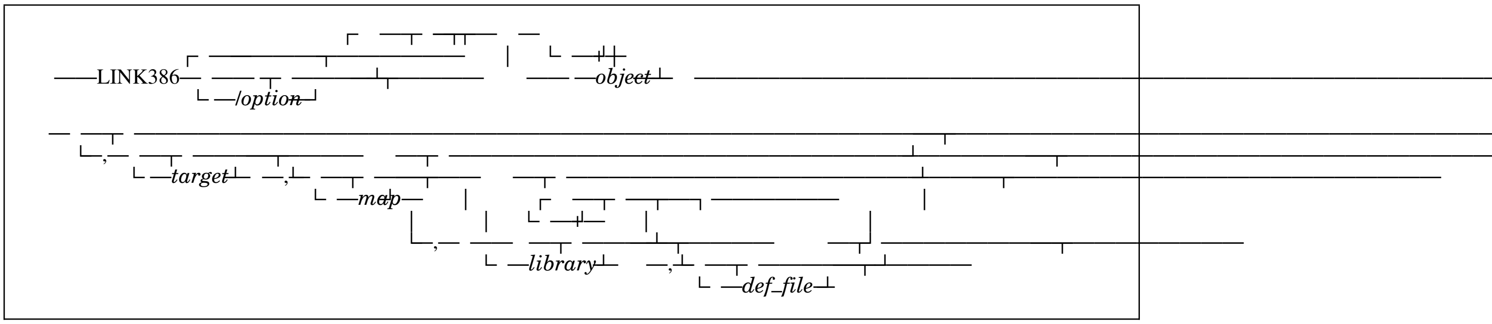
By default, the icc program invokes the linker automatically. It passes a number of default options and any options you specify using the /B compiler option.

If you want to link your program yourself, use the /C+ option with the icc command to specify compile only. You can then invoke the LINK386 program directly.

Important: If you are compiling C++ code that uses templates, you must invoke the linker through icc and not as a separate link step. You must also specify the /Tdp compiler option. This ensures that the compiler correctly resolves all template function definitions.

Linking Independently of the Compiler

The syntax for the LINK386 command is:



You can specify multiple options, objects, and libraries. If you specify multiple objects or libraries, separate them with a space or with a plus sign (+). At least one object is required, but the other parameters are optional. If you do not specify a parameter, the default is used. To skip a parameter and specify the following one, specify only the comma (,) as a placeholder.

The semicolon (;) ends the command line wherever it appears. For example, to link stan.obj using all the defaults, use the following command:

```
LINK386  stan.obj;
```

The linker command and options are described in detail in the Toolkit online *Tools Reference*.

When icc invokes the linker, it passes a number of linker options by default. If you link your program separately, you may want to specify these options:

/NOI
: Maintain case sensitivity for identifier names. If you link dynamically to the runtime libraries or if you use any C++ functions, you **must** use this option or your program will not load.

/BASE:65536 Specify the starting address of the program. Because the OS/2 operating system always loads executable programs at 64K, you can give the linker the address 65536 (or x1). If the linker knows where the program will be loaded, it can resolve relocation information at link time, resulting in a smaller and faster executable module. Use this option only when compiling .EXE files.

/ALIGN:16 Align segments on 16-byte boundaries inside the .EXE or .DLL file. This option reduces the size of the module, which in turn reduces load time.

/EXEPACK Pack the .EXE or .DLL file. This option reduces the size of the module, which in turn reduces load time.

Note: If you do not want to use the linker options passed by icc, you must link your program independently of icc.

If you use #pragma alloc_text or the /Nt option and plan to debug your code with the debugger, do not use the /PACKCODE linker option to group neighboring code segments. This option can interfere with the debugging of your program. It is not passed to the linker by default.

Creating Runtime DLLs

If your application uses functions from the C/C++ Tools libraries, you need to ensure the code for those libraries is always available to your application. You cannot ship the C/C++ Tools DLLs themselves with your application because of the product licensing agreement and because if more than one application included the C/C++ Tools DLLs, but at different levels, at least one application would be using the wrong level.

Binding Runtime Messages

If you are shipping your application to other users who do not have access to the library DLLs, you can use one of three methods to include the C/C++ Tools library code:

1. Statically bind every module to the library (.LIB) files.

   This method increases the size of your modules and slows the performance because the library environment has to be initialized for each module. Having multiple library environments also makes signal handling, file I/O, and other operations more complicated.

2. Use the DLL rename utility included with the C/C++ Tools product to rename the library DLLs and make the necessary changes in your executable files that call the DLLs.

   This method is described in detail in the READ.ME file.

3. Create your own runtime DLLs.

   This method provides one common runtime environment for your entire application. It also lets you apply changes to the runtime library without relinking your application, meaning that if the C/C++ Tools DLLs change, you need only rebuild your DLL.

For a description of how to build your own runtime DLL, see "Creating Your Own Runtime Library DLLs" on page 216. If you are using the subsystem libraries, see "Creating Your Own Subsystem Runtime Library DLLs" on page 313.

Binding Runtime Messages to Your Application

If you are shipping your application to other users, you will also need to bind the C/C++ Tools runtime messages to your application. Use the MSGBIND utility from the Toolkit to bind the messages.

The MSGBIND command has the following syntax:

```
──MSGBIND──input_file──────────────────────────────
```

The *input_file* identifies the executable file to which the messages are to be bound, the message file where the messages reside, and the message numbers to bind.

The C/C++ Tools runtime messages file are named DDE4.MSG (C runtime messages) and DDE46.MSG (Task Library runtime messages) and are located in the HELP directory under the main C/C++ Tools directory. For both of these files, the message numbers are the same as the message identifiers. The C runtime messages are numbered EDC5000 to EDC5167, and the Task Library messages are numbered EDC7001 to EDC7108.

For a detailed description and example of an *input_file*, as well as more information about MSGBIND, see the Toolkit online *Tools Reference*.

Creating Online Documentation

The Information Presentation Facility (IPF) is an Toolkit tool that you can use to create online information, to specify how it will appear on the screen, to connect various parts of the information, and to provide help information that can be requested by the user. IPF features include:

- A tagging language that formats text and provides ways to connect information and customize the information display.
- A compiler that creates online documents and help windows.
- A viewing program that displays formatted online documents (view).

The syntax for the IPF compiler command is:

```
──IPFC──source_file──┬──────────────────┬──────────────────
                     └─/──options───────┘
──┬──────────────────────────┬─────────────────────────────
  └─>──output_message_file───┘
```

Enabling help for applications requires programming code that communicates with IPF and with the PM APIs to display help windows.

For more information on creating help for applications, see the Toolkit online *Information Presentation Facility Reference*.

Using the Resource Compiler

The OS/2 Resource Compiler is a tool you can use to add application resources, such as strings, pointers, menus, bitmaps, and dialog templates, to a PM application. You can bind these resources directly to your executable file or build them into a DLL which is then called by the executable file at run time. You use OS/2 APIs to load the resources into the application.

The command to invoke the Resource Compiler has the following syntax:

```
        ┌───────────────────────┐
──RC──  ──┬──────────────────┴┬──────────  ──input_file──  ──┬──────────────
          └──/──option──┘                 └ ──output_file──┘
```

You can type RC alone at the command line to get help for the command.

Using the Resource Compiler, you can define and modify the resources for an executable file without affecting the file itself, meaning you do not need to recompile the file. You can create multiple customized applications by adding different resources to a single executable file.

The Resource Compiler is especially useful for international applications. You can define all language-dependent data, such as message strings, as resources. You can then modify the existing application for a different language by binding different resources to it.

It is often easier to create a resource DLL than to bind your resources into your executable file. With a DLL, the maintenance of resources is easier and there is less duplication of resources. You may even be able to use a common resource DLL for multiple applications. The steps for creating a resource DLL are described in "Creating Resource DLLs" on page 215.

Using the NMAKE Utility

You can use the Toolkit make utility NMAKE to invoke the compiler and linker and any other tools you use. The NMAKE program simplifies compiling programs that have more than one source file, especially when there have been changes to only some of the files. NMAKE saves time by performing actions on only the files that have changed, and on the files that incorporate or depend on the changed files.

NMAKE uses a *make file* to determine what actions are to be performed on which files. You can write your own make files, or, if you have the WorkFrame/2 product, you can use its Make File Creation utility to create your make files.

Make File Creation Restrictions: When you create a make file using the WorkFrame/2 Make File Creation utility, only the #include preprocessor directive is recognized. All other preprocessor directives (for example, #if, #define) are ignored.

If a #include directive is used in conjunction with other preprocessor directives, it may be interpreted in a different way than was intended. For example:

Given the following directives:

```
#define XXX "kim.h"
#include XXX
```

the Make File Creation utility cannot determine the name of the file to be included because it does not process the #define directive. In this situation, an error message is generated.

For the following example, no error message is generated, but the results may not be as expected:

```
#if XXX
#include "kim.h"
#else
#include "alex.h"
#endif
```

Because the conditional directives are ignored, the make file created will have a dependency on both kim.h and alex.h.

For more information on NMAKE, see the Toolkit documentation. For more information on the Make File Creation utility, see the WorkFrame/2 online help.

Part 3. Running Your Program

This part describes how to set environment variables for running your program, how to specify runtime options, and how to redirect standard input/output.

Running Your Program

Chapter 7. Setting Runtime Environment Variables

You can set the runtime environment for the C/C++ Tools compiler by using OS/2 environment variables. Most of these variables can be set from the command line, in your CONFIG.SYS file, or in a command file using the SET command, or from within your program using the _putenv function.

The functions that access these environment variables are not available when you are using the subsystem libraries. To access the environment variables when using the subsystem libraries, you must use OS/2 APIs. See the Toolkit online *PM Programming Reference* for more information about OS/2 APIs.

Note: You can put an optional semicolon at the end of the commands that set the environment variables so that you can later append values to the variables from the command line.

Some of the variables discussed in this chapter are also used at compile time. The compiler environment variables are described in "OS/2 Environment Variables for Compiling" on page 34. For more information on environment variables in general, see the OS/2 2.0 documentation.

PATH

The system, _exec, and _spawn functions use this environment variable to search for .EXE and .CMD files not in the current directory. You can set it by entering PATH as a command or using the SET command. For example, the following two commands are equivalent:

```
SET PATH=c:\ibmc;e:\ian;d:\steve
PATH=c:\ibmc;e:\ian;d:\steve
```

If you set the PATH variable in your CONFIG.SYS file, you must use the SET command.

You can specify one or more directories with this variable. Given the above example, the path searched would be the current directory, and then the directories c:\ibmc, e:\ian, and d:\steve, in that order.

For further information on the functions that use PATH, refer to the *C Library Reference*.

DPATH

This environment variable is used at run time to locate information to support the setlocale function. Also, if the runtime messages are not bound to the executable module, the program searches for them first in the current directory, and then in the directory or directories specified by the DPATH variable. (The list of C/C++ Tools message files can be found in Appendix E, "Component Files" on page 431.)

For example, given the following DPATH value:

```
DPATH=c:\kevin;d:\michel
```

the program would search the current directory, and then the c:\kevin and d:\michel directories, in that order.

The DPATH variable can be set by entering DPATH as a command or by using the SET command. If you set DPATH in your CONFIG.SYS file, you must use the SET command.

LIBPATH

If you link dynamically to the C/C++ Tools libraries, the operating system searches the directories specified by this environment variable to find .DLL files required by the program. The library DLLs and any user DLLs must be in one of the directories specified by the LIBPATH.

This variable can only be specified in the CONFIG.SYS file. For example:

```
LIBPATH=c:\cmlib;c:\ibmc\dll;c:\ibmc\lib
```

sets the DLL search path to the c:\cmlib, c:\ibmc\dll, and c:\ibmc\lib directories. LIBPATH **cannot** be specified using the SET command. For more information on DLLs, see Chapter 12, "Building Dynamic Link Libraries" on page 195. For a list of all C/C++ Tools DLLs, see Appendix E, "Component Files" on page 431.

TMP

The directory specified by this variable holds temporary files, such as those created using the tmpfile function. (See the *C Library Reference* for a description of tmpfile.) You must set the TMP variable to use the C/C++ Tools compiler.

Set the TMP variable with the SET command in the CONFIG.SYS file or on the command line. For example:

```
SET TMP=c:\ibmc\tmp
```

You can specify only one directory using the TMP variable.

TEMPMEM

Use this variable to control whether temporary files are created as memory files or as disk files. It can be set using the SET command in the CONFIG.SYS file or on the command line. For example:

```
SET TEMPMEM=on
```

If the value specified is on (in upper-, lower-, or mixed case), and you compile with the /Sv+ option, the temporary files will be created as memory files. If TEMPMEM is set to any other value, the temporary files will be disk files. If you do not compile with /Sv+, memory file support is not available and your program will end with an error when it tries to open a memory file.

If TEMPMEM is used by a program, its value must be set in the environment before the program starts. You cannot set it from within the program.

COMSPEC

The system function uses this variable to locate the command interpreter. When the OS/2 operating system is installed, the installation program sets the COMSPEC variable in the CONFIG.SYS file to the name and path of the command interpreter. To change the COMSPEC variable, use the SET command in CONFIG.SYS. For example:

```
SET COMSPEC=c:\mydir\mycmd.exe
```

sets the command interpreter as mycmd.exe in the c:\mydir directory.

For more information on the system function, refer to the *C Library Reference*.

TZ

This variable is used to describe the time zone information to be used by the locale. It is set using the SET command, and has the following format:

SET TZ=*SSS*[+|-]*nDDD*[,*sm*,*sw*,*sd*,*st*,*em*,*ew*,*ed*,*et*,*shift*]

The values for the TZ variable are defined below. The default values given are for the built-in "C" locale defined by the ANSI C standard.

Figure 11 (Page 1 of 2). TZ Environment Variable Parameters

Variable	Description	Default Value
SSS	Standard time zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EST
n	Difference (in hours) between the standard time zone and coordinated universal time (CUT), formerly Greenwich mean time (GMT). A positive number denotes time zones west of the Greenwich meridian, a negative number denotes time zones east of the Greenwich meridian.	5

Figure 11 (Page 2 of 2). TZ Environment Variable Parameters

Variable	**Description**	**Default Value**
DDD	Daylight saving time (DST) zone identifier. This must be three characters, must begin with a letter, and can contain spaces.	EDT
sm	Starting month (1 to 12) of DST.	4
sw	Starting week (-4 to 4) of DST.	1
sd	Starting day of DST. 0 to 6 if *sw* != 0 1 to 31 if *sw* = 0	0
st	Starting time (in seconds) of DST.	3600
em	Ending month (1 to 12) of DST.	10
ew	Ending week (-4 to 4) of DST.	-1
ed	Ending day of DST. 0 to 6 if *ew* != 0 1 to 31 if *ew* = 0	0
et	Ending time of DST (in seconds).	7200
shift	Amount of time change (in seconds).	3600

For example:

```
SET TZ=CST6CDT
```

sets the standard time zone to CST, the daylight saving time zone to CDT, and sets a difference of 6 hours between CST and CUT. It does not set any values for the start and end of daylight saving time.

When TZ is not present, the default is EST5EDT, the "C" locale value. When only the standard time zone is specified, the default value of *n* (difference in hours from GMT) is   instead of 5.

If you give values for any of *sm, sw, sd, st, em, ew, ed, et*, or *shift*, you must give values for all of them. If any of these values is not valid, the entire statement is considered not valid, and the time zone information is not changed.

The value of TZ can be accessed and changed by the _tzset function. See the *C Library Reference* for more information on _tzset.

Runtime Environment Variables

Chapter 8. Running Your Program

After you create an executable file, you can run your program. On the command line, enter the name of the executable file with or without the extension.

Note: If the extension is not .EXE, you must include the extension.

The OS/2 operating system uses the PATH environment variable to find executable files. You can run a program from any directory, as long as the executable program is either:

1. In your current working directory.
2. In one of the directories specified by the PATH environment variable.
3. Specified on the command line with a fully-qualified path name.

The runtime messages files (DDE4.MSG for the C runtime and DDE46.MSG for the C++ Task Library runtime) must also be either in your current working directory or in one of the directories specified by the DPATH environment variable.

The system function provided in the C/C++ Tools runtime library lets you run other programs and OS/2 commands from within a program. See the *C Library Reference* for more information on the system function.

Passing Data to a Program

To pass data to your program by way of the command line, give one or more arguments after the program name. Each argument must be separated from other arguments by one or more spaces or tab characters. You must enclose any arguments that include spaces, tab characters, double quotation marks, or redirection characters, in double quotation marks. For example:

```
hello 42 "de f" 16
```

Passing Data to a Program

This command runs the program named hello.exe and passes three arguments: 42, de f, and 16. The combined length of all arguments in the command (including the program name) cannot exceed the OS/2 maximum length for a command.

You can also use escape sequences within arguments. For example, to represent double quotation marks, precede the double quotation character with a backslash. To represent a backslash, use two backslashes in a row. For example, when you invoke the hello.exe program from the preceding example with this command:

```
hello "ABC\"" \"HELLO\\
```

the arguments passed to the program are ABC" and "HELLO\.

Declaring Arguments to main

To set up your program to receive the command-prompt data, declare arguments to main as:

```
int main(int argc, char   argv, char   envp)
```

By declaring these variables as arguments to main, you make them available as local variables. You need not declare all three arguments, but if you do, they must be in the order shown. To use the envp argument, you must declare argc and argv, even if you do not use them.

Each OS/2 command-line argument, regardless of its data type, is stored as a null-terminated string in an array of strings. The command is passed to the program as the argv array of strings. The number of arguments appearing at the command prompt is passed as the integer variable argc.

The first argument of any command is the name of the program to run. The program name is the first string stored at argv[ ]. Because you must always give a program name, the value of argc is at least 1.

The runtime initialization code stores the first argument after the program name at argv[1], the second at argv[2], and so on through the end of the arguments. The total number of arguments, including the program name, is stored in argc. argv[argc] is set to a NULL pointer.

You can also access the values of the individual arguments from within the program using argv. For example, to access the value of the last argument, use the expression argv[argc-1].

The third argument passed to main, envp, is a pointer to the environment table. You can use this pointer to access the value of the environment settings. (Note that the getenv function accomplishes the same task and is easier to use.) The envp argument is not available when you use the subsystem libraries.

The _putenv routine may change the location of the environment table in storage, depending on storage requirements; because of this, the value given to envp when you start to run your program might not be correct throughout the running of the program. The _putenv and getenv functions access the environment table correctly, even when its location changes. For more information about _putenv and getenv see the *C Library Reference*.

Expanding Global File-Name Arguments

You can use the OS/2 global file-name characters (or wildcard characters), the question mark (?) and asterisk (*), to specify the file-name and path-name arguments at the command prompt. To use them, you must link your program with the special routine contained in SETARGV.OBJ. This object file is included with the libraries in the LIB directory under the main C/C++ Tools directory. If you do not link your program with SETARGV.OBJ, the compiler treats the characters literally.

Global File-Name Arguments

SETARGV.OBJ expands the global file-name characters in the same manner that the OS/2 operating system does. (See the OS/2 *Master Help Index* for more information.) For example, when you link hello.obj with SETARGV.OBJ:

```
LINK386 /NOE hello SETARGV;
```

and run the resulting executable module hello.exe with this command:

```
hello .INC ABC? "XYZ?"
```

the SETARGV function expands the global file-name characters and causes all file names with the extension .INC in the current working directory to be passed as arguments to the hello program. Similarly, all file names beginning with ABC followed by any one character are passed as arguments. The file names are sorted in lexical order.

If the SETARGV function finds no matches for the global file-name arguments, for example, if no files have the extension .INC, the argument is passed literally.

Because the "XYZ?" argument is enclosed in quotation marks, the expansion of the global file-name character is suppressed, and the argument is passed literally as XYZ?.

WorkFrame/2 Considerations: If you have installed the IBM WorkFrame/2 product and you frequently use global file-name expansion, you can place the SETARGV.OBJ routine in the standard libraries you use. Then the routine is automatically linked with your program.

Use the WorkFrame/2 LIB utility to delete the module named SETUPARG from the library (the module name is the same in all C/C++ Tools libraries), and add the SETARGV module. When you replace SETUPARG with SETARGV, global file-name expansions are performed automatically on command-line arguments.

For more information on the LIB utility, see the online information for the WorkFrame/2 product.

Redirecting Standard Streams

A C or C++ program has standard streams associated with it. You do not have to open them because they are automatically set up by the runtime environment when you include <**stdio.h**>. The three standard streams are:

stdin
: The input device from which your program normally retrieves its data. For example, the library function getchar uses stdin.

stdout
: The output device to which your program normally directs its output. For example, the library function printf uses stdout.

stderr
: The output device to which your program directs its diagnostic messages.

The streams stdprn and stdaux are reserved for use by the OS/2 operating system and are not supported by the C/C++ Tools compiler.

On input and output operations requiring a file pointer, you can use stdin, stdout, or stderr in the same manner as you would a regular file pointer.

When a C++ program uses the I/O Stream library, the following predefined streams are also provided in addition to the standard streams:

cin
: The standard input stream.

cout
: The standard output stream.

cerr
: The standard error stream. Output to this stream is unit-buffered. Characters sent to this stream are flushed after each insertion operation.

clog
: Also the standard error stream. Output to this stream is fully buffered.

Redirecting Standard Streams

The cin stream is an istream_withassign object, and the other 3 streams are ostream_withassign objects. These streams and the classes they belong to are described in detail in the *Standard Class Library Reference*.

There may be times when you want to redirect a standard stream to a file. The following sections describe methods you can use for C and C++ programs.

Redirection from within a Program

To redirect C standard streams to a file from within your program, use the freopen library function. For example, to redirect your output to a file called pia.out instead of stdout, code the following statement in your program:

```
freopen("pia.out", "w", stdout);
```

For more information on freopen, refer to the *C Library Reference*.

You can reassign a C++ standard stream to another istream (cin only) or ostream object, or to a streambuf object, using the operator=. For example, to redirect your output to a file called michael.out, create michael.out as an ostream object, and assign cout to it:

```
#include <fstream.h>

int main(void)
{
     cout << "This is going to the standard output stream" << endl;

     ofstream outfile("michael.out");
     cout = outfile;
     cout << "This is going to michael.out file" << endl;

     return  ;
}
```

You could also assign cout to outfile.rdbuf() to perform the same redirection.

For more information on using C++ standard streams, see the *Standard Class Library Reference*.

Redirection from the Command Line

To redirect a C or C++ standard stream to a file from the command line, use the standard OS/2 redirection symbols.

For example, to run the program bill.exe, which has two required parameters XYZ and 123, and redirect the output from stdout to a file called bill.out, you would use the following command:

```
bill XYZ 123 > bill.out
```

You cannot use redirection from the command line for memory files.

You can also use the OS/2 file handles to redirect one standard stream to another. For example, to redirect stderr to stdout, you would use the command:

```
2 > &1
```

Refer to the OS/2 online *Master Help Index* for more information on redirection symbols.

Returning Values from main

The function main, like any other C function, returns a value. Its return value is an int value that is passed to the operating system as the return code of the program that has been run. You can check this return code with the IF ERRORLEVEL command in OS/2 batch files. See the OS/2 online *Command Reference* for more information on the IF ERRORLEVEL command.

To cause main to return a specific value to the operating system, use the return statement or the exit function to specify the value to be returned. For example, the statement

```
return 6;
```

returns the value 6. If you do not use either method, the return code is undefined.

For more information about main, see the *Online Language Reference*.

Part 4. Coding Your Program

This part describes different features of the C/C++ Tools compiler that you may want to use when you code your program, including the input and output methods, the support for multithread programs and dynamic link libraries, and ways to improve program performance and to reduce size.

Coding Your Program

Chapter 9. Input/Output Operations

This chapter describes input and output methods for the C/C++ Tools compiler. Note that no record level I/O is supported, including that described by the SAA definition.

Standard Streams

Three standard streams are associated with the C language, stdin, stdout, and stderr. In C++, when you use the I/O Stream Library, there are 4 additional C++ standard streams, cin, cout, cerr, and clog. All of the standard streams are described in "Redirecting Standard Streams" on page 143.

An OS/2 file handle is associated with each of the streams as follows:

File Handle	C Stream	C++ Stream
0	stdin	cin
1	stdout	cout
2	stderr	cerr, clog

Note: Both cerr and clog are standard error streams; cerr is unit-buffered and clog is fully buffered.

Note: The file handle and stream are not equivalent. For example, there may be a situation where file handle 2 is associated with a stream other than stderr, cerr or clog. Do not code your program in such a way that it is dependent on the association between the stream and the file handle.

The standard streams are not available when you are using the subsystem libraries.

The streams stdprn and stdaux are reserved for use by the OS/2 operating system and are not supported by the C/C++ Tools product.

Note: The C++ streams do not support the use of ddnames. See the *Standard Class Library Reference* for more information about the C++ streams.

Stream Processing

Input and output are mapped into logical data streams, either text or binary. The properties of the streams are more uniform than the properties of their input and output.

Text Streams

Text streams contain printable characters and control characters organized into lines. Each line consists of zero or more characters and ends with a new-line character (\n). A new-line character is not automatically appended to the end of the file.

The C/C++ Tools compiler may add, alter, or ignore some new-line characters during input or output so that they conform to the conventions for representing text in an OS/2 environment. Thus, there may not be a one-to-one correspondence between the characters in a stream and those in the external representation. See the example on page 152 for an example of the difference in representations.

Data read from a text stream is equal to the data that was written if it consists only of printable characters and the horizontal tab, new-line, vertical tab, and form-feed control characters.

On output, each new-line character is translated to a carriage-return character, followed by a line-feed character. On input, a carriage-return character followed by a line-feed character, or a line-feed character alone is converted to a new-line character.

If the last operation on the stream is a read, fflush discards the unread portion of the buffer. If the last operation on the stream is a write, fflush writes out the contents of the buffer. In either case, fflush clears the buffer.

The ftell, fseek, fgetpos, fsetpos, and rewind functions cannot be used to get or change the file position within character devices or OS/2 pipes.

The C standard streams are always in text mode at the start of your program. You can change the mode of a standard stream from text to binary without redirecting the stream by using the freopen function with no file name specified, for example:

```
fp = freopen("", "rb", stdin);
```

You can use the same method to change the mode from binary back to text. You cannot change the mode of a stream to anything other than text or binary, nor can you change the file type to something other than disk. No other parameters are allowed. Note that this method is included in the SAA C definition, but not in the ANSI C standard.

Control-Z Character

When a text stream is connected to a character device, such as the keyboard or an OS/2 pipe, the Ctrl-Z (\x1a) character is treated as an end-of-file indicator, regardless of where it appears in the stream.

If Ctrl-Z is the last character in a file, it is discarded when read. Similarly, when a file ending with a Ctrl-Z character is opened in append or update mode, the Ctrl-Z is discarded. The C/C++ Tools product does not automatically append a Ctrl-Z character to the end of text files which it writes. If you require a Ctrl-Z character at the end of your text files, you must write it out yourself.

This treatment of the Ctrl-Z character applies to text streams only. In binary streams, it is treated like any other character.

Binary Streams

A binary stream is a sequence of characters or data. The data is not altered on input or output, so the data read from a binary stream is equal to the data that was written.

If the last operation on the stream is a read, fflush discards the unread portion of the buffer. If the last operation on the stream is a write, fflush writes out the contents of the buffer. In either case, fflush clears the buffer.

The gets function reads the bytes from stdin up to and including the new-line character. It then replaces the new-line character with a null character (\ ).

The fgets function reads from a specified stream until it encounters the end of file, a new-line character, or until it has read $n - 1$ bytes (*n* is given as a parameter to fgets). If read, the new-line character is included in the string.

Differences between Storing Data as a Text or Binary Stream

If two streams are opened, one as a binary stream and the other as a text stream, and the same information is written to both, the contents of the streams may differ. In the following example of two streams of different types, the hexadecimal values of the resulting files, which show how the data is actually stored, are not the same.

```
#include <stdio.h>

int main(void)
{
   FILE  fp1,  fp2;
   char lineBin[15], lineTxt[15];
   int  x;

   fp1 = fopen("script.bin","wb");
   fprintf(fp1,"hello  world\n");

   fp2 = fopen("script.txt","w");
   fprintf(fp2,"hello  world\n");

   fclose(fp1);
   fclose(fp2);
```

Figure 12 (Part 1 of 2). Differences between Binary and Text Streams

```
    fp1 = fopen("script.bin","rb");

    / opening the text file as binary to suppress
    the conversion of internal data    /
    fp2 = fopen("script.txt","rb");

    fgets(lineBin, 15, fp1);
    fgets(lineTxt, 15, fp2);

    printf("Hex value of binary file = ");
    for (x= ; lineBin[x]; x++)
        printf("%.2x", (int)(lineBin[x]) );

    printf("\nHex value of text file     = ");
    for (x= ; lineTxt[x]; x++)
        printf("%.2x", (int)(lineTxt[x]) );

   printf("\n");

   fclose(fp1);
   fclose(fp2);

    / The expected output is:

       Hex value of binary file = 68656c6c6f2 776f726c64 a
       Hex value of text file     = 68656c6c6f2 776f726c64 d a    /
}
```

Figure 12 (Part 2 of 2). Differences between Binary and Text Streams

As the hexadecimal values of the file contents show in the binary stream (script.bin), the new-line character is converted to a line-feed (\ a), while in the text stream (script.txt), the new-line is converted to a carriage-return line-feed (\ d a).

Memory File Input/Output

When you compile with the /Sv+ option, the C/C++ Tools compiler supports files known as **memory files**. Memory files differ from the other file types only in that they are temporary files that reside in memory; you can write to and read from a memory file just like a disk file.

Using memory files can speed up the execution of your program because, under normal circumstances, there is no disk I/O when your program accesses these files. However, if your program is running in an environment where the operating system is swapping shared memory into and out of virtual memory on disk, you might not get faster execution when using memory files. This case is most likely to be true if your memory files are large.

Use fopen to create a memory file by:

Specifying type=memory. For example

```
stream = fopen("memfile.txt", "w, type=memory");
```

Using the SET DD: statement with the memory(y) option. For example

```
SET DD:MEMFILE=memfile.txt, memory(y)
fopen("DD:MEMFILE", "w");
```

The SET DD: statement specifies MEMFILE as a *data definition name* (ddname).

Notes:

1. You must specify the /Sh+ compiler option to use ddnames.
2. Ddnames are not supported for use with C++ standard streams.

Once a memory file has been created, it can be accessed by the module that created it as well as by any other function within the same process. The memory file remains accessible until the file is removed by the remove function or until the program has terminated.

A call to fopen that tries to open a file with the same name as an existing memory file accesses the memory file, even if you do not specify type=memory in the fopen call.

When using fopen to open a memory file in write or append mode, you must ensure that the file is not already open.

Memory File Restrictions and Considerations

| You must specify the /Sv+ option to use memory files.

Memory files are private to the process that created them. Redirection to memory files from the command line is not supported, and they cannot be shared with any other process, including child processes. Also, memory files cannot be shared through the system function.

Memory files do not undergo any conversion of the new-line character, meaning that data is not altered on input or output.

Memory files are unbuffered, and the blksize attribute is ignored. No validation is performed for the path or file name used.

Memory file names are case sensitive. For example, the file a.a is not the same memory file as A.A. In the following example,

```
fopen("A.A","w,type=memory");
remove("a.a");
```

the call to remove will not remove memory file A.A because the file name is in uppercase. Because memory files are always checked first, the function will look for memory file a.a, and if that file does not exist,
| it will remove the disk file a.a (or A.A, because disk files are not case
| sensitive).

Buffering

You can request that the temporary files created by the tmpfile function be either disk files or memory files. By default, tmpfile creates temporary files as disk files. To have temporary files created as memory files, set the TEMPMEM environment variable to ON:

```
SET TEMPMEM=on
```

The word on can be in any case. You must still specify the /Sv+ compiler option. For more information about TEMPMEM, see Chapter 7, "Setting Runtime Environment Variables" on page 133.

Buffering

The C/C++ Tools compiler uses buffers when it performs I/O operations to increase the efficiency of system-level I/O. The following buffering modes are used:

Unbuffered Characters are transmitted as soon as possible. This mode is also called unit buffered.

Line buffered Characters are transmitted as a block when a new-line character is encountered or when the buffer is filled.

Fully buffered Characters are transmitted as a block when the buffer is filled.

The buffering mode specifies the manner in which the buffer is flushed, if a buffer exists.

You can use the blksize= parameter with the fopen function or the blksize(*n*) parameter with a ddname to indicate the initial size of the buffer you want to allocate for the stream. Note that you must specify the /Sh+ compiler option to use ddnames.

If you do not specify a buffer size using fopen or a ddname, the default buffer size is 4096. The setvbuf and setbuf functions can be used to control buffering before any read or write operation to the stream. These functions must be specified for each stream. You cannot change the buffering mode after any operation on the file has occurred.

Fully-buffered mode is the default unless the stream is connected to a character device, in which case it is line-buffered.

To ensure data is transmitted to external storage as soon as possible, use the setbuf or setvbuf function to set the buffering mode to unbuffered.

Note: The C/C++ Tools product does not support pipes created using the DosCreateNmPipe API.

Opening Streams Using Data Definition Names

When you specify the /Sh+ compiler option, you can use the OS/2 SET command with a data definition name (ddname) as a parameter to specify the files to be opened by your program. You can also use the SET command to specify other file characteristics.

When you use the SET command with ddnames, you can change the files that are accessed by each run of your program without having to alter and recompile your source code.

Notes:

1. You cannot use ddnames with the C++ standard streams.
2. The maximum number of files that can be open at any time changes with the amount of memory available.

Specifying a ddname with the SET Command

To specify a ddname, the SET command has the following syntax:

SET DD:*DDNAME*=*filename*[,option, option...]

where:

DDNAME Is the ddname as specified in the source code. The ddname **must** be in uppercase.

filename Is the name of the file that will be opened by fopen.

No white-space characters are allowed between the DD and the equal sign.

Setting File Characteristics with ddnames

For example, you could open the file sample.txt in two ways:

By putting the name of the file directly into your source code.

```
FILE  stream;
stream=fopen("sample.txt", "r");
```

By using a ddname in the fopen call and the SET command to specify the file you want your program to open.

```
FILE  stream;
stream=fopen("DD:DATAFILE", "r");
```

Before you run your program, use the SET command:

```
SET DD:DATAFILE=c:\sample.txt
```

When the program runs, it will open the file c:\sample.txt. If you want the same program to use the file c:\test.txt the next time it runs, use the following SET command:

```
SET DD:DATAFILE=c:\test.txt
```

The SET command can be issued before your program is executed by entering it on the command line, including it in a batch file, or putting it into the CONFIG.SYS file. You can also use the _putenv function from within the program to set the ddname. For example:

```
_putenv("DD:DATAFILE=sample.txt, writethru(y)");
```

See the *C Library Reference* for a description of _putenv.

Describing File Characteristics Using Data Definition Names

The options that you can use when defining ddnames allow you to specify the characteristics of the file your program opens. You can specify the options in any order, in upper- or lowercase. If you specify an option more than once, only the last one takes effect. If an option is not valid, fopen fails and errno is set accordingly.

Setting File Characteristics with ddnames

You can use the following options when specifying a ddname:

blksize(*n* **)**
: The size in bytes of the block of data moved between the disk and the program. The maximum size is 32760 for fixed block files and 32756 for variable block files. Larger values can improve the efficiency of disk access by lowering the number of times the disk must be accessed. Typically, values below 512 increase I/O time, and values above 8K do not show improvement.

lrecl(*n* **)**
: The size in bytes of one record (logical record length). If the value specified is larger than the value of blksize, the lrecl value is ignored.

recfm(f | v | fb | vb)[2]
: Specifies whether the files are fixed or variable block size.

- **f** The block size is fixed.
- **v** The block size is variable.
- **fb** The block size is fixed and is an even multiple of the logical record length.
- **vb** The block size is variable and is an even multiple of the logical record length.

share (read | none | all)
: Specifies the file sharing.

- **read** The file can be shared for read access. Other processes can read from the file, but not write to it.
- **none** The file cannot be shared. No other process can get access to the file (exclusive access).
- **all** Allows the file to be shared for both read and write access. Other processes can both read from and write to the file.

2 The default values for these options are underlined.

writethru(n | y)
Determines whether to force the writing of OS/2 buffers.

n Turns off forced writes to the file. The system is not forced to write the internal buffer to permanent storage before control is returned to the application.

y Forces the system to write to permanent storage before control is returned to the application. The directory is updated after every write operation.

Use writethru(y) if data must be written to the disk before your program continues. This can help make data recovery easier should a program interruption occur.

Note: When writethru(y) is specified, file output will be noticeably slower.

memory(n | y)
Specifies whether a file will exist in permanent storage or in memory.

n Specifies that the file will exist in permanent storage.

y Specifies that the file will exist only in memory. The system uses only the OS/2 file name. All other parameters, such as a path, are ignored. You must specify the /Sv+ option to enable memory files.

fopen **Defaults**

A call to fopen has the following defaults:

blksize The default buffer size of 4K (4096 bytes) is used.

share(read) The file can be shared for read access. Other processes can read from the file, but not write to it.

writethru(n) The file is opened with no forced writes to permanent storage.

Full buffering is used unless the stream is connected to a character device, in which case it is line-buffered.

For more information on fopen, refer to the *C Library Reference*.

Precedence of File Characteristics

You can describe your data both within the program, by fopen, and outside it, by ddname, but you do not always need to do so. There are advantages to describing the characteristics of your data in only one place.

Opening a file by ddname may require the merging of the information internal and external to the program. In the case of a conflict, the characteristics described by using fopen override those described using a ddname. For example, given the following ddname statement and fopen command:

```
SET DD:ROGER=danny.c, memory(n)
stream = fopen("DD:ROGER", "w, type=memory")
```

the file danny.c will be opened as a memory file.

Options you specify in the application program using _putenv take precedence over any that are set in the ddname environment string.

Closing Files

The fclose function is used to close a file. On normal program termination, the compiler automatically closes all files and flushes all buffers. When a program ends abnormally, all files are closed but the buffers are not flushed.

Input/Output Restrictions

The following restrictions apply to input/output operations:

- Seeking within character devices and OS/2 piped files is not allowed.
- Seek operations past the end of the file are not allowed for text files. For binary files that are opened using any of w, w+, wb+, w+b, or wb, a seek past the end of the file will result in a new end-of-file position and nulls will be written between the old end-of-file position and the new one.

Note: When you open a file in append mode, the file pointer is positioned at the end of file.

I/O Considerations when You Use Presentation Manager

Standard I/O functions such as printf write to OS/2 file handle 1, which is the default destination of stdout and cout. Unless you redirect the output and messages, they are not visible through the Presentation Manager (PM) interface.

There are two ways to display the output sent to stdout or cout depending on whether you want to see the output while the program is running or after it has finished:

1. To see the output while the program is running, you must pipe the output stream to some other program that reads input and displays it using PM calls. For example, to pipe the output from junko.exe to the program display (which uses PM calls to write to the screen), use the following command:

   ```
   junko | display
   ```

2. To view the output after the program has finished, redirect the output stream to a file. You can do this from a command line, for example:

   ```
   junko > file.out
   ```

   or from within the file using the freopen function:

   ```
   freopen("file.out", "w", stdout);
   ```

To send output from a C/C++ Tools application directly to a PM window, you must use PM calls.

All error messages during run time go to OS/2 file handle 2, which is the default destination of stderr, cerr, and clog. Like output to file handle 1, these messages are not visible through the PM interface. To see the error messages, you must redirect the error stream to a file.

For more details on redirecting output, see "Redirecting Standard Streams" on page 143.

Chapter 10. Optimizing Your Program

This chapter describes different ways to improve your program's performance (optimize for speed), as well as how to decrease the size of your executable module (optimize for size). Note that in some cases, optimizing for one quality means the other will suffer.

The recommendations in this chapter provide guidelines only. To obtain the best results for either performance or module size, you may have to experiment with the techniques suggested. The benefits to your program may vary depending on your code and on the opportunities for optimization available to the compiler.

Improving Program Performance

This section lists the methods you can use to improve the speed of your program.

Choosing Compiler Options

The following list names the compiler options that can improve performance. It also describes what each option does to cause the improvement. Note that none of these options is the default.

Option	**Effect**
/Gf+	Generates code for fast floating-point operations.
/Gi+	Generates code for fast integer operations.
/Gx+	For C++ programs only, suppresses generation of exception handling code.
/G[3|4|5]	Optimize for the 386 (/G3), 486 (/G4), or Pentium (/G5) microprocessor. Use the appropriate option for the processor you are using or plan to use. If you do not know what processor your application will be run on, use the /G3 option.
/O+	Turns on optimization. The C/C++ Tools compiler always optimizes for speed. Specifying /O+ also causes /Op+ (enable optimizations involving the stack pointer) and /Os+ (invoke the instruction scheduler) to be specified.

/Oi+ Inlines user functions.

/Ol+ Passes code through the intermediate code linker. Using the intermediate linker can result in better optimized code. For best results, use the /Gu+ option also to specify that unreferenced data is not used by external functions. See "Using the Intermediate Code Linker" on page 52 for more information about the intermediate linker.

/Om- Does not limit the working set size of the compiler. The compiler is then able to inline more user code.

The following options improve the performance of your code by preventing the generation of objects or information that can degrade performance. Note that these are set by default:

Option	Effect
/Gh-	Does not generate profiler hooks.
/Gr-	Generates code to run in the usual operating system environment. If you use /Gr+, the code generated runs at ring , and the performance suffers. Some code, such as device drivers, must run at ring .
/Gv-	Does not save and restore the DS and ES registers for external function calls.
/Gw-	Does not generate an FWAIT instruction after each floating-point load instruction.
/Ti-	Does not generate debug information.
/Ts-	Does not generate code to allow the debugger to maintain the call stack.

If your program has only one thread, use the /Gs+ option to disable stack probes. (/Gs- is the default.) Because the stack of the first thread is always fully committed, stack probes are not necessary in single-thread programs. If your program has multiple threads, stack probes serve a useful purpose and you should probably use them. See "Controlling Stack Allocation and Stack Probes" on page 67 for more information about stack probes.

If you link your executable files in a separate link step, specify the /BASE:65536 linker option to tell the linker your executable file will be loaded at 64K. The linker can then resolve a number of references that would otherwise have to be resolved by the loader at load time and by the pager as the program runs. When you use icc to link your program, it specifies this option for you by default.

Note: Do not use the /BASE:65536 for DLLs.

Specifying Linker Options

Using the following linker options can lead to improved performance. Note that when icc invokes the linker, it passes these options by default:

/BASE:65536 Specify the starting address of the program. Because the OS/2 operating system always loads executable programs at 64K, you can give the linker the address 65536 (or x1). If the linker knows where the program will be loaded, it can resolve relocation information at link time, resulting in a smaller and faster executable module.

Note: Only .EXE files are loaded at address 65536. When you compile DLLs, specify a load address that is comparatively large (for example, x8) and unique for each DLL (to prevent the code from overlapping between DLLs). If the value does not meet these criteria, your program will still run, but will not gain any improvement in performance from the /BASE option.

/ALIGN:16 Align segments on 16-byte boundaries inside the .EXE or .DLL file. This option reduces the size of the module, which in turn reduces load time.

/EXEPACK Pack the .EXE or .DLL file. This option reduces the size of the module, which in turn reduces load time.

Choosing Libraries

Your choice of runtime libraries can affect the performance of your code:

- Use the subsystem library whenever possible. Because there is no runtime environment for this library, its load and initialization times are faster than the other libraries.
- Use the single-thread library for single-thread programs. The multithread library involves extra overhead.
- If your application has multiple executable modules and DLLs, create and use a common version of a runtime library DLL. See "Creating Your Own Runtime Library DLLs" on page 216 for information on how to create your own runtime library DLL.

Allocating and Managing Memory

The following list describes ways to improve performance through better memory allocation and management:

- If you allocate a lot of dynamic storage for a specific function, use the _alloca function. Because _alloca allocates from the stack instead of the heap, the storage is automatically freed when the function ends. In some cases however, using _alloca can detract from performance. It causes the function that calls it to chain the EBP register, which creates more code in the function prolog and also eliminates EBP for use as a general-purpose register. For this reason, if _alloca is not called often in your function, use one of the other memory allocation functions.
- You can use either malloc or DosAllocMem to allocate storage. In general, DosAllocMem is faster, but you must do your own heap management and you cannot use realloc to reallocate the memory. malloc manages the heap for you and the storage it returns can be reallocated with realloc. malloc is also more portable than DosAllocMem.
- When you use malloc, keep in mind that the amount of storage allocated is actually the amount you specify plus an additional 16 bytes that is used internally by the memory allocation functions.

When you copy data into storage allocated by calloc, malloc, or realloc, copy it to the same boundaries on which the compiler would align them. In particular, aligning double precision floating-point variables and arrays on 8-byte boundaries can greatly improve performance on the 486 and Pentium microprocessors. For more information about the mapping of data, see "Data Mapping" on page 389.

When you declare or define structures or C++ classes, take into account the alignment of data types. Declare the largest members first to reduce wasted space between members and reduce the number of boundaries the compiler must cross. The alignment is especially important if you pack your structure or class.

Periodically after freeing or reallocating storage several time, call _heapmin to release the unused storage to the operating system and reduce the working set of your program. A reduced working set causes less swapping of memory to disk, which in turn results in better performance. Experiment to determine how often you should call _heapmin.

Using Strings and String Manipulation Functions

The handling of string operations can also affect the performance of your program:

Use #pragma strings (readonly) to make your strings read-only. If you use the intrinsic string functions, the compiler can better optimize them if it knows that any string literals they are operating on will not be changed.

When you store strings into storage allocated by malloc, align the start of the string on a doubleword boundary. This alignment allows the best performance of the string intrinsic functions. The compiler performs this alignment for all strings it allocates.

Keep track of the length of your strings. If you know the length of your string, you can use memcpy instead of strcpy The memcpy function is faster because it does not have to search for the end of the string.

- Avoid using strtok. Because this function is very general, you can probably write a function more specific to your application and get better performance.

Performing Input and Output

There are a number of ways to improve your program's performance of input and output:

- Use binary streams instead of text streams. In binary streams, data is not changed on input or output.
- Use the low-level I/O functions, such as _open and _close. These functions are faster and more specific to the application than the stream I/O functions like fopen and fclose. Note that you must provide your own buffering for the low-level functions.
- If you do your own I/O buffering, make the buffer a multiple of 4K, which is the size of a page. Because malloc adds an extra 16 bytes of storage, allocating storage in a multiple of the page size actually results in more pages being allocated than required. Instead, use DosAllocMem to allocate this storage for the buffer.
- If you are using a file as a temporary file and performing frequent read or write operations on it, use memory files. Because memory files operate on the memory of the system, I/O operations can be performed more quickly on memory files than on disk files. Note that to use memory files you must specify the /Sv+ option.
- Instead of scanf and fscanf, use fgets to read in a string, and then use one of atoi, atol, atof, or _atold to convert it to the appropriate format.
- Use sprintf only for complex formatting. For simpler formatting, such as string concatenation, use a more specific string function.
- When reading input, read in a whole line at once rather than one character at a time.

Designing and Calling Functions

Whether you are writing a function or calling a library function, there are a few things you should keep in mind:

- Fully prototype all functions. A full prototype gives the compiler and optimizer complete information about the types of the parameters. As a result, promotions from unwidened types to widened types are not required and the compiler does not need to emit eyecatcher instructions for the function. (See “Eyecatchers” on page 241 for a description of eyecatchers.)
- When designing a function, place the most used parameters in the left-most position in the function prototype. The left-most parameters have a better chance of being stored in a register.
- Avoid passing structures or unions as function parameters or returning a structure or union. Passing aggregates requires the compiler to copy and store many values. Pass or return a pointer to the structure or union instead.
- If near the end of your function, you call another function and pass it the same parameters that were passed to your function, put them in the same order in the function prototypes. The compiler can then reuse the storage that the parameters are in and does not have to generate code to reorder them.
- Use the intrinsic and built-in functions, which include string manipulation, floating-point, and trigonometric functions. Intrinsic functions require less overhead and are faster than a function call, and often allow the compiler to perform better optimization.

- Be careful when using intrinsic functions in loops. Many intrinsic functions use multiple registers. Some of the registers are specific and cannot be changed. In the loop, the number of values to be placed in registers increases while the number of registers is limited. As a result, temporary values such as loop induction variables and results of intermediate calculations often cannot be stored in registers, which slows your program performance.

  In general, you will encounter this problem with the intrinsic string functions rather than the floating-point functions. Often if the arguments to the string function are in global registers, this problem does not occur.

- Use recursion only where necessary. Because recursion involves building a stack frame, an iterative solution is always faster than a recursive one.

Other Coding Techniques

The following list describes other techniques you can use to improve performance:

- Minimize the use of external (extern) variables to reduce aliasing and improve optimization.

- Avoid taking the address of local variables. If you use a local variable as a temporary variable and must take its address, avoid reusing the temporary variable. Taking the address of a local variable inhibits optimizations that would otherwise be done on calculations involving that variable.

- Avoid using short int values, except in aggregates. Because all integer arithmetic is done on long values, using short values causes extra conversions to be performed.

- If you do division or modulo arithmetic by a divisor that is a power of 2, if possible, make the dividend unsigned to produce better code.

- Use #pragma alloc_text and #pragma data_seg to group code and data respectively, to improve the locality of reference. Variables and functions that are used at the same time are stored together, and might fit on a single page that can be used and then discarded. You can use EXTRA to determine which functions should be grouped together.

Use _Optlink linkage wherever possible. Keep _Optlink as your default linkage and use linkage keywords to change the linkage for specific functions.

If a loop body has a constant number of iterations, use constants in the loop condition to improve optimization. For example, the statement for (i= ; i<4; i++) can be better optimized than for (i= ; i<x; i++).

Use the intermediate code linker to improve optimization. See “Using the Intermediate Code Linker” on page 52 for information about the intermediate linker.

Inline your functions selectively. Inlined functions require less overhead and are generally faster than a function call. The best candidates for inlining are small functions that are called frequently. Large functions and functions that are called rarely may not be good candidates for inlining.

For best results, use EXTRA to decide which functions you should inline and qualify them with the _Inline keyword (or inline for C++ files). Using automatic inlining (specifying /Oi with a value) is not as effective. Using the intermediate code linker with user inlining can improve your program performance even more.

Certain coding practices will slow down your performance. Only use them if you need to. They are often necessary, but you should be aware that they will affect your program's performance:

Calling 16-bit code. The compiler performs a number of conversions to allow interaction between 32-bit and 16-bit code.

Using the setjmp and longjmp functions. These functions involve storing and restoring the state of the thread.

Using #pragma handler. This #pragma causes code to be generated to register and deregister an exception handler for a function.

Using unprototyped variable argument functions. Due to the nature of the _Optlink calling convention, unprototyped variable-length argument lists make performance slower. Prototype all of your functions. Also, if you use variable argument functions, use the _System calling convention.

C++-Specific Considerations

The following performance hints apply only to C++ programs:

- Because C++ objects are often allocated from the heap and have a limited scope, memory usage in C++ programs affects performance more than in C programs. To improve memory usage and performance:
  - Tailor your own new and delete operators.
  - Allocate memory for a class before it is required.
  - Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an object manager. Each time you create an instance of an object, you pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.
  - Avoid copying large complex objects.
- When you use the Collection class library to create classes, use a high level of abstraction. After you establish the type of access to your class, you can create more specific implementations. This can result in improved performance with minimal code change.
- Use virtual functions only when they are necessary. Virtual functions are usually compiled to be indirect calls, which are slower than direct calls.
- Use try blocks for exception handling only when necessary because they can inhibit optimization.
- Use the /Gx+ option to suppress the generation of exception handling code in programs where it is not needed. Unless you specify this option, some exception handling code is generated even for programs that do not use catch or try blocks.
- Avoid using overloaded operators to perform arithmetic operations on user-defined types. The compiler cannot perform the same optimizations for objects as it can for simple types.
- Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point.

The result is two objects that point to the same contained object. A deep copy, however, copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on. A simple assignment using an overloaded operator can generate many lines of code.

Reduce the indirect interaction between classes. For example, use friend classes to reduce the overhead of access methods.

When you define structures or data members within a class, define the largest data types first to align them on the doubleword boundary.

Reducing Program Size

This section lists the methods you can use to decrease the size of your executable module.

Choosing Compiler Options

The following list names the compiler options to use to make your executable module smaller. Unless noted, these options are not set by default.

/Gd+ Links dynamically to the runtime library. If you link statically, code for all the runtime functions you call is included in your executable module.

/Gf+ Generates code for fast floating-point execution and eliminates certain conversions.

Note: Code produced using /Gf+ does not conform to ANSI or IEEE standards.

/Gh- Does not generate profiler hooks which would increase module size. This is the default.

/Gi+ Generates code for fast integer execution and eliminates certain conversions.

/Gv- Does not save and restore the DS and ES registers for external function calls. This is the default.

/Gw- Does not generate an FWAIT instruction after each floating-point load instruction. This is the default.

Reducing Program Size

/Gx+ For C++ programs only, suppresses generation of exception handling code.

/G3 Optimizes for the 386 processor. This is the default. Optimizing for the 486 or Pentium microprocessor generates extra code. Code compiled with /G3 runs on a 486 or Pentium microprocessor.

/O+ Turns on optimization.

/Oi- Does not inline user functions. Inlining reduces overhead but increases module size. When /O- is specified, this is the default. When /O+ is specified, /Oi+ becomes the default.

/Ol+ Passes code through the intermediate code linker. The intermediate linker removes unused variables and sorts external data to provide maximal packing. For best results, use the /Gu+ option to specify that defined data is not used by external functions. See "Using the Intermediate Code Linker" on page 52 for more information about the intermediate linker.

/Sh- Does not include ddname support. This is the default.

/Sv- Does not include memory file support in the library. This is the default.

/Ti- Does not generate debug or EXTRA information, which would increase module size. This is the default.

/Ts- Does not generate code to allow the debugger to maintain the call stack, which would increase module size. This is the default.

/Tx- Provides only the exception message and address when an exception occurs instead of a complete machine-state dump. This is the default.

If you link your program in a separate link step, specify the /ALIGN:1 linker option to align segments on 1-byte boundaries. The default alignment is 4-byte boundaries. You should also specify the /EXEPACK linker option, which compresses repeated byte patterns within pages of data.

Using Libraries and Library Functions

Your choice of libraries and of library functions affect the size of your code:

- Use the subsystem library whenever possible. This library has no runtime environment, meaning the initialization, termination, and exception handling code is not included. It also includes fewer library functions than the standard library.
- Use the low-level I/O functions. Note that you must provide your own buffering for these functions.
- Disable the intrinsic functions. Certain string manipulation, floating-point, and trigonometric functions are inlined by default. (See "Intrinsic Functions" on page 383 for a list of these functions.) To disable the inlining, parenthesize the function call, for example:

```
(strlen)("ian");
```

  Note that for most of the floating-point intrinsics, this recommendation does not apply because the inlined code is probably smaller than a generated call instruction.

Other Coding Techniques

The following list describes other ways you can make your modules smaller:

- If you do not use the argc and argv arguments to main, create a dummy _setuparg function that contains no code.
- Avoid assigning structures. Instead, use memcpy to copy the structure.
- If you do not use the intermediate code linker, arrange your own external data to minimize gaps in alignment.
- When you declare or define structures or C++ classes, take into account the alignment of data types. Declare the largest members first to reduce wasted space between members.
- If you must use the intrinsic string manipulation functions, use #pragma strings(readonly) to make your strings read-only.

Optimizing for Both Speed and Size

This section describes how to make your executable module both faster and smaller. Note that when you optimize for both speed and size, the gains you make on either quality are less than if you were optimizing for one quality alone.

In general, follow the guidelines in "Improving Program Performance" on page 165, except where they are contraindicated in "Reducing Program Size" on page 175. For example, intrinsic functions may improve performance, but they also increase the size of your module, so you may want to avoid using them.

Choosing Compiler Options

The following compiler options have a positive effect on both performance and code size:

/O+ Turns optimization on.

/Ol+ Passes codes through the intermediate code linker. For best results, use the /Gu+ option also.

/Gf+ Generates code for fast floating-point execution and reduces floating-point conversions.

/Gh- Does not generate profiler hooks.

/Gi+ Generates code for fast integer execution and reduces integer conversions.

/Gw- Does not generate a FWAIT instruction after each floating-point load instruction.

/Ti- Does not generate debug information.

/Ts- Does not generate code to allow the debugger to maintain the call stack.

/Tx- Provides only the exception message and address when an exception occurs instead of a complete machine-state dump.

If you link your program separately, use the /BASE:65536 and /EXEPACK linker options.

Chapter 11. Creating Multithread Programs

This chapter describes how to use the C/C++ Tools compiler to create multithread programs and discusses restrictions of the multithread environment. It also describes the sample multithread program included with the C/C++ Tools product that you may have installed. For the sample code and instructions on how to compile and run the sample program, see "Sample Multithread Program" on page 194.

Multithread programming is a feature of the OS/2 operating system. The C/C++ Tools compiler supports multithread programming with:

- Code generation and linking options. (See "Compiling and Linking Multithread Programs" on page 193 for more information.)
- Multithread libraries. (See "Libraries for Multithread Programs" on page 180 for more information.)

No multithread support is available in the subsystem libraries.

What Is a Multithread Program?

A multithread program is a program whose functions are divided among several threads. While a *process* is an executing application and the resources it uses, a *thread* is the smallest unit of execution within a process. Other than its stack and registers, a thread owns no resources; it uses those of its parent process. This chapter discusses only threads and refers to processes only for contrast.

Multithread programs allow more complex processing than single-thread programs. In a single-thread program, all operations are performed synchronously. That is, one operation begins when the preceding one has finished. In a multithread program, many threads execute at the same time, and the operations are performed concurrently.

Although threads within a process share the same address space and files, each thread runs as an independent entity and is not affected by the control flow of any other thread in the process. Because a function from any thread can perform any task, such as input or output, threads are well suited to concurrent programs that share data.

Libraries for Multithread Programs

The C/C++ Tools compiler has two standard libraries that provide library functions for use in multithread programs. The DDE4MBS.LIB library is a statically linked multithread library, and DDE4MBSI.LIB is an import multithread library, with the addresses of the functions contained in C/C++ Tools DLLs.

| The C++ Standard class libraries are not all available for multithread programs. The Task library is single-thread only because of the nature of the applications it generates. The Complex Mathematics library is available for both single- and multithread programs. The single-thread Complex library is COMPLEX.LIB, while the multithread version is COMPLEXM.LIB. The C++ I/O Stream library is built into the C/C++ Tools single-thread and multithread runtime libraries. The User Interface class library also offers a Thread class that is an encapsulation of the OS/2 APIs for multithread programming. You can use this class in your multithread programs to :

- Set thread priority
- Set thread attributes
- Do a reference count for objects dispatched on a thread so they are automatically deleted when the thread ends
- Dispatch a member function of a C++ object on a separate thread
- Control other aspects of your threads.

For a description of the Thread class and how to use it, see the *User Interface Class Library Reference*.

Thread Control

The multithread libraries provide two functions, _beginthread and _endthread, to create new threads and end them. These functions are described in detail in the *C Library Reference*. The C/C++ Tools compiler does not limit the number of threads you can create, but the OS/2 operating system does. For more information on the number of threads allowed, see the online *OS/2 Programming Reference*. The C/C++ Tools product also provides the global variable _threadid that identifies your current thread, and the function _threadstore that gives you a private thread pointer to which you can assign any thread-specific data structure.

You can also create threads with the DosCreateThread API. If you use DosCreateThread, you must use a #pragma handler directive for the thread function to ensure correct exception handling. You should also call _fpreset from the new thread to ensure the floating-point control word is set correctly for the thread. Although you can use DosExit to end threads created with DosCreateThread, you should use _endthread to ensure that the necessary cleanup of the environment is done.

Note: The function that is to run on the thread created by DosCreateThread must have _System linkage. If you need to start a new thread for a function with any other type of linkage, you must use _beginthread.

You should use _beginthread to create any threads that call C/C++ Tools library functions. When the thread is started, the library environment performs certain initializations that ensure resources and data are handled correctly between threads. Threads created by the DosCreateThread API do not have access to the resource management facilities or to C/C++ Tools exception handling. When you use _beginthread, the _endthread function is called automatically when the thread ends.

Using the Multithread Libraries

When you use the multithread libraries, you must consider a number of things that do not apply to the single-thread libraries. Because many library functions share data and other resources, the access to these resources must be serialized (limited to one thread at a time) to prevent functions from interfering with each other. Other functions do not require serialization of access but have other restrictions, or affect all threads running within a process. Global variables and error handling are also affected by the multithread environment.

Reentrant Functions

Reentrant functions can be suspended at any point and reentered, after which they can return to that same point to resume processing, with no adverse effects. Because these functions use only local variables, they cannot interfere with each other. Access to these functions is not serialized.

The following functions are reentrant:

abs	_fstat	localtime	_stat	strupr
acos	_ftime	log	strcat	strxfrm
asctime	_fullpath	log1	strchr	_swab
asin	gamma	_lrotl	strcmp	tan
assert	_gcvt	_lrotr	strcmpi	tanh
atan	_getcwd	_lsearch	strcoll	time
atan2	_getdcwd	_ltoa	strcpy	_toascii
atof	_getdrive	_makepath	strcspn	tolower
atoi	_getpid	mblen	_strdate	_tolower
atol	gmtime	mbstowcs	strerror	toupper
atold	hypot	mbtowc	_strerror	_toupper
bsearch	isalnum	memccpy	strftime	_tzset
_cabs	isalpha	memchr	stricmp	_ultoa
ceil	_isascii	memcmp	strlen	_utime
_chdir	iscntrl	memcpy	strlwr	vsprintf
_chdrive	isdigit	memicmp	strncat	_wait
clock	isgraph	memmove	strncmp	wcscat
cos	islower	memset	strncpy	wcschr
cosh	isprint	_mkdir	strnicmp	wcscmp
ctime	ispunct	mktime	strnset	wcscpy
_cwait	isspace	modf	strpbrk	wcscspn
difftime	isupper	pow	strrchr	wcslen
div	isxdigit	qsort	strrev	wcsncat
_ecvt	_itoa	_rmdir	strset	wcsncmp
erf	_j	_rotl	strspn	wcsncpy
erfc	_j1	_rotr	strstr	wcspbrk
exp	_jn	sin	_strtime	wcsrchr
fabs	labs	sinh	strtok	wcsspn
_fcvt	ldexp	_splitpath	strtod	wcstombs
floor	ldiv	sprintf	strtol	wctomb
fmod	_lfind	sqrt	strtold	_y
_freemod	_loadmod	sscanf	strtoul	_y1
frexp				_yn

Reentrant Functions

All functions in the C++ Complex Mathematics Library are fully reentrant. The I/O Stream Library functions are nonreentrant.

Although the reentrant functions do not require serialization of data access, there is an important exception: if you pass a pointer as a parameter, the function may no longer be reentrant and may therefore require that access is serialized.

The program in Figure 13 provides an example of unserialized data access in a multithread program. The example uses the strcpy function on the same array in two different threads. The strcpy function does not serialize access to data that it is passed as a parameter. It is therefore possible that string A could end up containing half of the string from function f1 and half of the string from f2.

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

char A[2 ] ;
int   f1_done =   ;
int   f2_done =   ;

void _Optlink f1 (void  argument)
{
    strcpy(A,"123456789 ");
     f1_done = 1 ;
}

void _Optlink f2 ( void   argument)
{
    strcpy(A,"abcdefghij");
     f2_done = 1 ;
}
```

Figure 13 (Part 1 of 2). Example of Unserialized I/O

```
int main(void)
{
    char  holder[8 ];

    _beginthread(f1,NULL,4 96,NULL);
    _beginthread(f2,NULL,4 96,NULL);

     while (1)    /  Break only when both threads are done.  /
    {
          printf("Press <enter> to continue.\n");
         gets(holder);
          if (f1_done && f2_done)
              break  ;
          printf("The threads are still executing! \n");
    }

     printf("A is %s.\n",A);
    return   ;
}
```

Figure 13 (Part 2 of 2). Example of Unserialized I/O

Nonreentrant Functions

The remaining C/C++ Tools library functions access data or resources that are common to every thread in the process, such as files, environment variables, or I/O resources. To prevent any interference between these functions, each function uses *semaphores* to serialize access to data and resources. A semaphore is a mechanism provided by the OS/2 operating system specifically for this purpose. Semaphores are described in detail in the Toolkit online *PM Reference*.

Operations involving file handles and standard I/O streams are serialized so multiple threads can send output to the same stream without intermingling the output.

Example of Serialized I/O

If thread1 and thread2 execute the calls in the example below, the output could appear in several different ways, but it will never be garbled as shown at the end of the example.

```
#include <stdio.h>

int done_1 =  ;
int done_2 =  ;

void _Optlink thread1(void)
{
    fprintf(stderr,"This is thread 1\n");
    fprintf(stderr,"More from 1\n");
    done_1 = 1;
}

void _Optlink thread2(void)
{
    fprintf(stderr,"This is thread 2\n");
    fprintf(stderr,"More from 2\n");
    done_2 = 1;
}
```

Figure 14 (Part 1 of 2). Example of Serialized I/O

```
int main(void)
{
     _beginthread(thread1, NULL, 4 96, NULL);
     _beginthread(thread2, NULL, 4 96, NULL);

    while  (1)
    {
          if (done_1 && done_2)
               break;
    }
    return   ;
}

/  Possible output could be:

               This is thread 1
               This is thread 2
               More from 1
               More from 2
or
               This is thread 1
               More from 1
               This is thread 2
               More from 2
or
               This is thread 1
               This is thread 2
               More from 2
               More from 1

     The output will never look like this:

               This is This is thrthread 1
              ead  2
               More froMore m 2
              from  1                                            /
```

Figure 14 (Part 2 of 2). Example of Serialized I/O

Several nonreentrant functions have specific restrictions:

- The getc, getchar, putc, and putchar file I/O operations are implemented as macros in the single-thread C libraries. In the multithread libraries, they are redefined as functions to implement any necessary serialization of resources.
- Use the _fcloseall function only after all file I/O has been completed.
- When you use printf or vprintf and the subsystem libraries, you must provide the necessary serialization for stdout yourself.

| The functions in the C++ I/O Stream Library are also nonreentrant. To use these I/O Stream objects in a multithread environment, you must provide your own serialization either using the OS/2 semaphore APIs or the IResourceLock, IPrivateResource, and ISharedResource classes from the User Interface class library.

Process Control Functions

The process termination functions abort, exit, and _exit end all threads within the process, not just the thread that calls the termination function. In general, you should allow only thread 1 to terminate a process, and only after all other threads have ended. Note that it is not always possible in a signal or exception handler for only thread 1 to terminate processes.

Note: A routine that resides in a DLL must **not** terminate the process, except in the case of a critical error.

Signal Handling in Multithread Programs

Signal handling, as described in Chapter 18, "Signal and OS/2 Exception Handling" on page 317, also applies to the multithread environment. The default handling of signals is usually either to terminate the program or to ignore the signal. Special-purpose signal handling, however, can be complicated in the multithread environment.

Signal handlers are registered independently on each thread. For example, if thread 1 calls signal as follows:

```
signal(SIGFPE, handlerfunc);
```

the handler handlerfunc is registered for thread 1 only. Any other threads are handled using the defaults.

A signal is always handled on the thread that generated it, except for SIGBREAK, SIGINT, and SIGTERM. These three signals are handled on the thread that generated them only if they were raised using the raise function. If they were raised by an exception, they will be handled on thread 1.

For more information and examples on handling signals, refer to Chapter 18, "Signal and OS/2 Exception Handling" on page 317.

Global Data and Variables

Data and variables that are global or shared between threads, such as errno and _environ, are implemented differently in the multithread libraries to prevent interference among functions that access or change their values. The global variables are handled in one of two ways: either the variable is made a per-thread variable, or access to the variable is serialized.

Per-Thread Global Variables

A per-thread global variable has a name that is common to all threads, but its value is specific to each thread. The value of the global variable may be different for each thread in the process.

The variables errno and _doserrno, which are used to return errors from library functions, are implemented as per-thread global variables. If these variables were not set on a per-thread basis, functions in multiple threads would overwrite each other's error codes. Use errno and _doserrno in the same manner as you would for a single thread program.

For example, the following program shows how the value of errno is unique to each thread. Although an error occurs in the thread openProc, the value of errno is because it is checked from the main thread.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

int done =  ;

void _Optlink openProc(void  argument)
{
    FILE  filePointer ;
    filePointer = fopen("C:\\OS2","w");
    printf("openProc, errno = %d\n",errno);
    done = 1;
}
```

Figure 15 (Part 1 of 2). Example of a Per-Thread Variable

```
int main(void)
{
    char  holder[8 ];

     errno =   ;
    _beginthread(openProc,NULL,4 96,NULL)  ;

     while (1)    /  Break only when the thread is done.  /
    {
          printf("Press <enter> to continue.\n");
         gets(holder);
         if  (done)
              break  ;
          printf("The thread is still executing! \n");
    }

     printf("Main program, errno = %d.\n",errno);
    return   ;

     /  The expected output is:

          Press <enter> to continue.
          openProc, errno = 6

          Main program, errno =  .           /

}
```

Figure 15 (Part 2 of 2). Example of a Per-Thread Variable

Signal handlers are also unique for each thread, and are registered on a per-thread basis.

The buffer to be passed to the longjmp function is allocated on a per-thread basis. When you call longjmp, the buffer you pass to it must have been created by a call to setjmp on the same thread. If the buffer was not created on the same thread, the process will terminate.

Global Variables in Multithread Programs

The internal buffers used by asctime, ctime, gmtime, and localtime are also allocated on a per-thread basis. That is, these functions return addresses of buffers that are specific to the thread from where the function was called.

There is one seed per thread for generating random numbers with the rand and srand functions to keep the pseudorandom numbers generated in each thread independent of other threads. Each thread starts with the same seed (1); that is, each thread gets the same sequence of pseudorandom numbers unless the seed is changed by a call to srand.

Global Variables Requiring Serialization

These global variables containing environment strings should be treated as read-only data:

```
int  _daylight;
long  _timezone;
char   _tzname;

char  _osmajor;
char  _osminor;
char  _osmode;

char    _environ;
```

The environment strings are copied from the OS/2 environment when a program starts. This procedure is the same in multithread and single thread programs. Because all threads share the environment strings, any changes made to the strings by one thread affects the environment accessed by the other threads.

To ensure that access to the environment variables is serialized, use _putenv to set the environment variables. Each thread can then call getenv to obtain a copy of the environment strings and copy the string to a private data buffer so that any later changes to the environment by _putenv will not affect it. If the thread must always access the latest version of the environment strings, it must call getenv each time. The _putenv and getenv functions are described in the *C Library Reference*.

Using Common Variables

User variables that are referenced by multiple threads should have the attribute volatile to ensure that all changes to the value of the variable are performed promptly by the compiler. For example, because of the way the compiler optimizes code, the following example may not work as intended when compiled with the /O+ option:

```
static int common_var;

/    code executing in thread 1    /

     common_var =  ;
           ...
     common_var = 1;
           ...
     common_var = 2;

/    code executing in thread 2    /
           ...
    switch  (common_var)
    {
        case   :
               ...
           break;
        case  1:
               ...
           break;
        default:
               ...
           break;
     }
```

When using optimization, the compiler may not immediately store the value 1 for the variable common_var in thread 1. If it determines that common_var is not accessed by this code until after the value 2 is stored, it may never store the value 1. Thread 2 therefore does not necessarily access the true value of common_var.

Declaring a variable as volatile indicates to the compiler that references to the variable have side effects, or that the variable may change in ways the compiler cannot determine. Optimization will not eliminate any action involving the volatile variable, and changes to the value of the variable are then stored immediately.

Compiling and Linking Multithread Programs

When you compile your multithread program, you must specify that you want to use the multithread libraries described in "Libraries for Multithread Programs" on page 180. Because threads share data, the operating system and library functions must ensure that only one thread is reading or writing data at one time. The multithread libraries provide this support. (You can use these libraries for single-thread programs, but the multithread support causes unnecessary overhead.)

To indicate that you want the multithread libraries, specify the /Gm+ compiler option. For example:

```
icc /Gm+ mymulti.c
```

Conversely, the /Gm- option, which is the default, specifies explicitly to use the single-thread version of the library.

If you intend to compile your source code into separate modules and then link them into one executable program file, you must compile each module using the /Gm+ option and ensure that the multithread libraries are used when you link them. You cannot mix modules that have been compiled with /Gm+ with modules compiled using /Gm-.

You can use either static (/Gd-) or dynamic (/Gd+) linking with multithread programs.

Sample Multithread Program

The SAMPLE02 sample program provides an example of a multithread program. It creates one thread for each numerical argument passed to it. Each thread then prints a message the number of times specified by the argument.

If you installed the sample programs, the files for SAMPLE02 are found in the SAMPLES\SAMPLE02 directory under the main C/C++ Tools directory. Two make files that build the sample are also provided, MAKE 2S for static linking and MAKE 2D for dynamic linking.

Note: You must have the Toolkit installed in order to use the make files.

To compile and link SAMPLE 2.C, at the prompt in the SAMPLES\SAMPLE02 directory, use NMAKE with the appropriate make file. For example:

```
NMAKE all /f MAKE 2D
```

To compile and link the program yourself, use the following command:

Command	Description
icc /Gm SAMPLE 2.C	Compiles and links SAMPLE 2.C using default options and the multithread library.

To run the program, type SAMPLE 2 followed by any number of numerical arguments. For example:

```
SAMPLE 2 2 4 1
```

Chapter 12. Building Dynamic Link Libraries

Dynamic linking is the process of resolving external references using dynamic link libraries (DLLs) at runtime. You can dynamically link with the supplied C runtime DLLs, as well as with your own DLLs.

There are four different types of DLLs that can be created with the C/C++ Tools compiler:

- User DLLs that use the regular C/C++ Tools runtime libraries. These DLLs can be linked to the C/C++ Tools libraries either statically or dynamically.
- User DLLs that use the C/C++ Tools subsystem libraries and have no runtime environment. These DLLs contain only those functions provided in the subsystem libraries and possibly some built-in functions. They can be linked to the C/C++ Tools libraries either statically or dynamically. Refer to Chapter 17, "Developing Subsystems" on page 303 for information on building subsystem DLLs.
- Runtime library DLLs, such as those shipped with the C/C++ Tools product. "Creating Your Own Runtime Library DLLs" on page 216 describes how to build your own library DLLs to ship with your application.
- Resource DLLs that contain no code but contain one or more resources, such as menus or icons, that are used by PM programs. You can create these DLLs using the Resource Compiler from the Toolkit. See "Creating Resource DLLs" on page 215 for information on how to create resource DLLs.

This chapter describes the following steps for creating and using a dynamic link library:

1. Creating the source files for a DLL
2. Creating a module definition file (.DEF) for the DLL
3. Compiling the source files and linking the resulting object files to build a .DLL file
4. Letting external modules know what is in the DLL, either by creating an import library file (.LIB) for the DLL, or by writing a module definition file to be used when linking the external module.

This chapter also gives additional information on how to create your own DLL initialization and termination function, your own library DLLs, and your own resource DLLs.

An example is provided at the end of each section to illustrate that section. The examples shown are from the sample program SAMPLE 3, which is supplied with the C/C++ Tools product. For information on how to compile, link, and run the sample program, see "Sample Program to Build a DLL" on page 207.

Creating DLL Source Files

To build a DLL, you must first create source files containing the data and/or functions that you want to include in your DLL. No special file extension is required for DLL source files. The source code can be written in C or C++.

Each function that you want to *export* from the DLL (that is, a function that you plan to call from other executable modules or DLLs) must be an external function, either by default or by being qualified with the extern keyword. linker will not find your function references and will generate errors.

If your DLL and the modules that access it do not dynamically link to the same runtime DLL, you must use the #pragma handler directive to ensure exceptions are handled properly within your DLL. Use #pragma handler at the entry point of each DLL function to register the library exception handler _Exception. On exit from the function, code will also be generated to deregister _Exception.

Note: You need to explicitly register the exception handler only for the functions that will be exported from the DLL. For more information on #pragma handler, see the *Online Language Reference*. For information on exception handling, see Chapter 18, "Signal and OS/2 Exception Handling" on page 317.

Example of a DLL Source File

The file SAMPLE03.C is the source file for the DLL used in the SAMPLE03 sample program. If you installed the sample programs, this file is found in the SAMPLES\SAMPLE03 directory under the main C/C++ Tools directory.

The source file contains the code for:

- Three sorting functions: bubble, insertion, and selection
- Two static functions, swap and compare, that are called by the sorting functions
- A function, list, that lists the contents of an array.

For instructions on how to compile, link, and run the sample program, see "Sample Program to Build a DLL" on page 207.

Initializing and Terminating the DLL Environment

The initialization and termination entry point for a DLL is the _DLL_InitTerm function. When each new process gains access to the DLL, this function initializes the necessary environment for the DLL, including storage, semaphores, and variables. When each process frees its access to the DLL, the _DLL_InitTerm function terminates the DLL environment created for that process.

The default _DLL_InitTerm function supplied by the C/C++ Tools compiler performs the actions required to initialize and terminate the runtime environment. It is called automatically when you link to the DLL.

If you require additional initialization or termination actions for your runtime environment, you will need to write your own _DLL_InitTerm function. For more information, see "Writing Your Own _DLL_InitTerm Function" on page 209. A sample _DLL_InitTerm function is included for the SAMPLE03 program. (See "Example of a User-Created _DLL_InitTerm Function" on page 211.)

Note: The _DLL_InitTerm function provided in the subsystem library differs from the runtime version. See "Building a Subsystem DLL" on page 306 for more information about building subsystem DLLs.

Creating a Module Definition File

A module definition (.DEF) file is a plain text file that describes the names, attributes, exports, imports, and other characteristics of an application or dynamic link library. You must use a module definition file when you create any OS/2 DLL.

Example of a Module Definition File

The .DEF file for the SAMPLE03 program is shown here to illustrate the most common statements used in a module definition file to build DLLs. For a complete description of module definition files, refer to the Toolkit online *Tools Reference* for the LINK386 program.

```
LIBRARY SAMPLE 3 INITINSTANCE TERMINSTANCE
PROTMODE
DATA MULTIPLE NONSHARED READWRITE LOADONCALL
CODE LOADONCALL
EXPORTS
   nSize       ; array size
   pArray      ; pointer to base of array of ints
   nSwaps      ; number of swaps required to sort the array
   nCompares ; number of comparisons required to sort the array
   list        ; array listing function
   bubble      ; bubble sort function
   insertion ; insertion sort function
```

Figure 16. SAMPLE03.DEF - DLL Module Definition File

The module statements specified in the .DEF file are as follows:

LIBRARY SAMPLE 3 INITINSTANCE TERMINSTANCE
: This statement identifies the executable file as a dynamic link library and specifies that SAMPLE 3 is the name of the DLL. It also uses the following attributes to specify when the _DLL_InitTerm function will be called:

  INITINSTANCE
  : The function is called the first time the DLL is loaded for each process that accesses the DLL. The alternative is INITGLOBAL; the function is called only the first time the DLL is loaded. INITGLOBAL is the default.

  TERMINSTANCE
  : The function is called the last time the DLL is freed for each process that accesses the DLL. The alternative is TERMGLOBAL; the function is called only the final time the DLL is freed. TERMGLOBAL is the default.

PROTMODE
: This statement specifies that the DLL can be run in protected (OS/2) mode only.

DATA MULTIPLE READWRITE LOADONCALL
This statement defines the default attributes for data segments within the DLL. The attributes are:

MULTIPLE
MULTIPLE specifies that there is a unique copy of the data segment for each process. The alternative is SINGLE; there is only one data segment for all processes to share. SINGLE is the default.

READWRITE
READWRITE means that you can read from or write to the data segment. The alternative is READONLY; you can only read from the data segment. READWRITE is the default.

LOADONCALL
LOADONCALL means that the data segment is loaded into memory when it is first accessed. The alternative is PRELOAD; the data segment will be loaded as soon as a process accesses the DLL. LOADONCALL is the default and is recommended over PRELOAD because it is much faster.

See "Defining Code and Data Segments" on page 201 for information on defining your own data segments.

CODE LOADONCALL
This statement defines the default attributes for code segments within the DLL. LOADONCALL means that the code segment is loaded when it is first accessed. The alternative to LOADONCALL is PRELOAD; the code segment is loaded as soon as a process accesses the DLL. LOADONCALL is the default. See "Defining Code and Data Segments" on page 201 for information on defining your own code segments.

EXPORTS
: This statement defines the names of the functions and variables to be exported to other runtime modules. Following the EXPORTS keyword are the export definitions, which are simply the names of the functions and variables that you want to export. Each name must be entered on a separate line. See "Defining Functions to be Exported" for more information on exporting functions.

Defining Code and Data Segments

In the .DEF file shown, all data and code segments are given the same attributes. If you want to specify different attributes for different sets of data or code, you can use the #pragma data_seg and #pragma alloc_text directives, or the /Nd and /Nt compiler options, to define your own data and code segments. You can then list the segments in the .DEF file under the heading SEGMENTS, and specify attributes for each. For example:

```
SEGMENTS
    mydata SHARED READONLY
   mycode  PRELOAD
```

Any segments that you do not specify under SEGMENTS are given the attributes specified by the DATA or CODE statement, depending on the type of segment.

For more information about #pragma data_seg and #pragma alloc_text, see the *Online Language Reference*. The /Nd and /Nt options are described under "Code Generation Options" on page 111.

Defining Functions to be Exported

When you export a function from a DLL, you make it available to programs that call the DLL. If you do not export a function, it can only be used within the DLL itself.

To export a function, list its name under the EXPORTS keyword in the .DEF file as described on page 201. Note that if your DLL is written in C++, you must specify the *mangled* or encoded name of the function. For a description of how to encode your function names, see "Demangling (Decoding) C++ Function Names" on page 386.

You can also use #pragma export or the _Export keyword to specify that a function is to be exported. For example, in SAMPLE3.C, the function selection is declared to be exported by a #pragma export directive. The #pragma directive also allows you to specify the name the exported function will have outside of the DLL and its ordinal number. When you use the keyword or #pragma directive for C++ functions, use the normal function name, not the encoded name.

If you use #pragma export or _Export to export your function, you may still need to provide an EXPORTS entry for that function. If your function has any of the following default characteristics

- Has shared data
- Has no I/O privileges
- Is not resident

it does not require an EXPORTS entry. If your function has characteristics other than the defaults, the only way you can specify them is with an EXPORTS entry in your .DEF file.

For more information about _Export and #pragma export, see the *Online Language Reference*.

C++ Considerations: For C++ DLLs, ensure that you export all member functions that are required. Some functions that are inlined or exported may use private or protected members that must then also be exported. In addition, you should export all static data members. If you do not export the static data members of a particular class, users of that class cannot debug their code because the reference to the static data members cannot be resolved.

Compiling and Linking Your DLL

To compile your source files to create a DLL, use the /Ge- compiler option. You may also want to use the /C+ option to compile your files without linking them, and then link them separately.

You must also specify the runtime libraries you want to use:

- Single-thread (/Gm-) or multithread (/Gm+). See Chapter 11, "Creating Multithread Programs" on page 179 for information on multithread libraries.
- Statically linked (/Gd-) or dynamically linked (/Gd+). See "Static and Dynamic Linking" on page 64 for more information on static and dynamic linking.

  Note: The method of linking used for the runtime libraries is independent of the module type you create; you can statically link the runtime functions in a dynamic link library.

For more information on compiler options, see "Specifying Compiler Options" on page 71.

When you use icc to compile and link your DLL, you must specify on the command line all the DLL source files followed by the module definition file. The name of the first source file (without the .C extension) is used as the name of the DLL.

For example, to compile and link the files mydlla.c and mydllb.c, using the mydll.def module definition file, use the command:

```
icc /Ge- mydlla.c mydllb.c mydll.def
```

Note: The /Ge- option tells the compiler you are building a DLL, rather than an executable file. The options to indicate the single-thread library (/Gm-) and to link the runtime libraries statically (/Gd-) are the defaults.

The resulting DLL will be called mydlla.dll.

Compiling and Linking Your DLL

If you are compiling and linking separately, you must give the following information to the LINK386 linker:

- The compiled object (.OBJ) files for the DLL
- The name to give the DLL
- The C libraries to use
- The name of the module definition file.

Note: The compiler includes information in the object files on the C libraries you indicated by the compiler options that control code generation (see 111). These libraries are automatically used at link time. You do not need to specify libraries on the linker command line unless you want to override the ones you chose at compile time.

For example, the following commands:

- Compile the source files mydlla.c and mydllb.c
- Link the resulting object files with the single-thread, statically linked C libraries, using the definition file mydll.def

to create the DLL finaldll.dll:

```
icc /C+ /Ge- mydlla.c mydllb.c
LINK386   /ALIGN:16 /EXEPACK /NOI mydlla.obj mydllb.obj,finaldll.dll,,,mydll.def;
```

You could use icc to both compile and invoke the linker for you with the following command:

```
icc /Ge- /Fefinal.dll mydlla.c mydllb.c mydll.def
```

If your DLL contains C++ code that uses templates, you must use icc to invoke the linker to ensure the templates are corrctly resolved. You must also specify the /Tdp compiler option.

Note: The icc command passes the linker options /NOI, /ALIGN:16, and /EXEPACK to the linker by default. The /NOI option preserves the case of external names, /ALIGN:16 option causes segments to be aligned on 16-byte boundaries, and /EXEPACK compresses repeated byte patterns within pages of data.

Using Your DLL

Write the source files that are to access your DLL as if the functions and/or variables are to be statically linked at compile time. Then when you link the program, you must inform the linker that some function and/or variable references are to a DLL and will be resolved at run time. There are two ways to communicate this information to the linker:

1. Use the IMPLIB utility (from the Toolkit) to create a library file with all the information that the linker needs about the DLL. The IMPLIB utility uses a module definition file to create an import library (.LIB) file for the DLL. When you link an executable module, the linker uses this import library to resolve external references to the DLL.

   If your DLL contains any C++ code that uses templates, you must always use it by means of an import library to ensure that the names you use when you instantiate the template are resolved correctly.

   If you invoke the linker directly, give the name of the import library where you normally specify library names. For example:

   ```
   LINK386 /NOI mymain.obj,,,finaldll.lib;
   ```

   If you invoke the linker through the icc command, you must put the name of the import library in the compiler invocation string. For example:

   ```
   icc mymain.c finaldll.lib
   ```

   See the Toolkit online *Tools Reference* for more information on IMPLIB.

   Note: The import libraries for the C/C++ Tools runtime DLLs have been supplied with the compiler.

2. Construct a module definition file for the accessing module that is being linked. The definition file specifies which variables and names will be obtained from a DLL at run time, and in which DLLs these items will be found. In general, import libraries are easier to use and maintain than module definition files.

Using Your DLL

Note: To make functions in a DLL available to other programs, the name of those functions must have been exported (using #pragma export or the _Export keyword in the source file, or with an EXPORT entry in the .DEF file) when the DLL was linked. Also, all DLLs must be in a directory listed in the LIBPATH environment variable (as described in Chapter 7, "Setting Runtime Environment Variables" on page 133).

Sample Definition File for an Executable Module

The following figure shows the module definition file used for the main program in the sample program SAMPLE 3.

```
NAME MAIN 3 WINDOWCOMPAT

IMPORTS
   SAMPLE 3.nSize
   SAMPLE 3.pArray
   SAMPLE 3.nSwaps
   SAMPLE 3.nCompares
   SAMPLE 3.list
   SAMPLE 3.bubble
   SAMPLE 3.insertion
```

Figure 17. MAIN03.DEF - Definition File for an Executable Module

The statements given are as follows:

NAME MAIN 3 WINDOWCOMPAT
: The NAME statement assigns the name MAIN 3 to the program being defined. If no name is given, the name of the executable module (without the .EXE extension) is used. WINDOWCOMPAT specifies that the program is compatible with the PM environment. The alternatives are NOTWINDOWCOMPAT, which means the program is not compatible with the PM environment, or WINDOWAPI, which means the program uses PM APIs.

IMPORTS
This statement defines the names of functions and variables to be imported for the program. Following the IMPORTS keyword are the import definitions. Each definition consists of the name of the DLL where the function or variable is to be found, followed by a period, followed by the name of the function or variable. Each definition must be entered on a separate line.

You can also use #pragma import to specify that a function is imported from a DLL. For example, in MAIN3.C, the function selection is declared to be imported using #pragma import. You can use the #pragma directive to import the function by name or by ordinal number. For a detailed description of #pragma import, see the *Online Language Reference*.

Sample Program to Build a DLL

The sample program SAMPLE03 shows how to build and use a DLL that contains three different sorting functions. These functions keep track of the number of swap and compare operations required to do the sorting.

The files for the sample program are:

SAMPLE3.C — The source file for the DLL, described in "Example of a DLL Source File" on page 197.

INITTERM.C — The _DLL_InitTerm function, shown in "Example of a User-Created _DLL_InitTerm Function" on page 211.

SAMPLE3.DEF — The module definition file for the DLL, shown in "Creating a Module Definition File" on page 198.

MAIN3.DEF — The module definition file for the executable, shown in "Sample Definition File for an Executable Module" on page 206.

SAMPLE3.H — The user include file.

MAIN3.C — The main program.

Sample Program to Build a DLL

If you installed the sample programs, these files are found in the SAMPLES\SAMPLE03 directory under the main C/C++ Tools directory. Two make files that build the sample are also provided, MAKE 3S for static linking and MAKE 3D for dynamic linking.

Note: You must have the Toolkit installed to use the make files.

To compile and link this sample program, at the prompt in the SAMPLES\SAMPLE03 directory, use NMAKE with the appropriate make file. For example:

```
nmake all /f MAKE 3S
```

To compile and link the program yourself, use the following commands:

Command	Description
icc /Ge- /B"/NOE" /DSTATIC_LINK SAMPLE 3.C INITTERM.C SAMPLE 3.DEF	Compiles and links SAMPLE 3.C using default options and Creating a DLL (/Ge-) Passing the /NOE option to the linker Defining STATIC_LINK.
Note: The /NOE linker option tells the linker to ignore the extended library information found in the object files. The linker then uses the version of _DLL_InitTerm that you provide instead of the one from the C/C++ Tools runtime library.	
icc MAIN 3.C MAIN 3.DEF	Compiles MAIN 3.C using default options.

To run the program, enter MAIN 3.

Writing Your Own _DLL_InitTerm Function

If your DLL requires initialization or termination actions in addition to the actions performed for the runtime environment, you will need to create your own _DLL_InitTerm function. The prototype for the _DLL_InitTerm function is:

```
unsigned long _System _DLL_InitTerm(unsigned long modhandle,
                                     unsigned long flag);
```

If the value of the *flag* parameter is 0, the DLL environment is initialized. If the value of the *flag* parameter is 1, the DLL environment is ended.

The *modhandle* parameter is the module handle assigned by the operating system for this DLL. The module handle can be used as a parameter to various OS/2 API calls. For example, DosQueryModuleName can be used to return the fully qualified path name of the DLL, which tells you where the DLL was loaded from.

The return code from _DLL_InitTerm tells the loader if the initialization or termination was performed successfully. If the call was successful, _DLL_InitTerm returns a nonzero value. A return code of indicates that the function failed. If a failure is indicated, the loader will not load the program that is accessing the DLL.

Because it is called by the operating system loader, the _DLL_InitTerm function must be compiled using _System linkage.

Note: A _DLL_InitTerm function for a subsystem DLL has the same prototype, but the content of the function is different because there is no runtime environment to initialize or terminate. For an example of a _DLL_InitTerm function for a subsystem DLL, see "Example of a Subsystem _DLL_InitTerm Function" on page 308.

Initializing the Environment

Before you can call any C/C++ Tools library functions, you must first initialize the runtime environment. Use the function _CRT_init, which is provided in the runtime libraries. The prototype for this function is:

```
int  _CRT_init(void);
```

If the runtime environment is successfully initialized, _CRT_init returns . A return code of -1 indicates an error. If an error occurs, an error message is written to file handle 2, which is the usual destination of stderr.

If your DLL contains C++ code, you must also call __ctordtorInit after _CRT_init to ensure that static constructors and destructors are initialized properly. The prototype for __ctordtorInit is:

```
void  __ctordtorInit(void);
```

Note: If you are providing your own version of the _matherr function to be used in your DLL, you must call the _exception_dllinit function after the runtime environment is initialized. Calling this function ensures that the proper _matherr function will be called during exception handling. The prototype for this function is:

```
void _Optlink _exception_dllinit( int ( )(struct exception  ) );
```

The parameter required is the address of your _matherr function.

Terminating the Environment

If your DLL is statically linked, you must use the _CRT_term function to correctly terminate the C runtime environment. The _CRT_term function is provided in the C/C++ Tools runtime libraries. It has the following prototype:

```
void  _CRT_term(void);
```

If your DLL contains C++ code, you must also call __ctordtorTerm before you call _CRT_term to ensure that static constructors and destructors are terminated correctly. The prototype for __ctordtorTerm is:

```
void  __ctordtorTerm(void);
```

Once you have called _CRT_term, you cannot call any other library functions.

If your DLL is dynamically linked, you cannot call library functions in the termination section of your _DLL_InitTerm function. If your termination routine requires calling library functions, you must register the termination routine with DosExitList. Note that all DosExitList routines are called before DLL termination routines.

Example of a User-Created _DLL_InitTerm Function

The following figure shows the _DLL_InitTerm function for the sample program SAMPLE 3.

```
#define   INCL_DOSMODULEMGR
#define   INCL_DOSPROCESS
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

/  _CRT_init is the C run-time environment initialization function.                /
/  It will return   to indicate success and -1 to indicate failure.                /

int _CRT_init(void);
#ifdef    STATIC_LINK

/  _CRT_term is the C run-time environment termination function.                   /
/  It only needs to be called when the C run-time functions are statically     /
/  linked.                                                                         /

void _CRT_term(void);
#else
```

Figure 18 (Part 1 of 4). INITTERM.C - _DLL_InitTerm Function for SAMPLE03

```
/  A clean up routine registered with DosExitList must be used if runtime      /
/  calls are required and the runtime is dynamically linked.    This will      /
/  guarantee that this clean up routine is run before the library DLL is      /
/  terminated.                                                                  /

static void _System cleanup(ULONG ulReason);
#endif
size_t nSize;
int  pArray;

/  _DLL_InitTerm is the function that gets called by the operating system      /
/  loader when it loads and frees this DLL for each process that accesses      /
/  this DLL.    However, it only gets called the first time the DLL is loaded  /
/  and the last time it is freed for a particular process.    The system       /
/  linkage convention MUST be used because the operating system loader is      /
/  calling this function.                                                       /

unsigned long _System _DLL_InitTerm(unsigned long hModule, unsigned long
                                                    ulFlag)
{
    size_t  i;
    APIRET  rc;
    char  namebuf[CCHMAXPATH];

     /  If ulFlag is zero then the DLL is being loaded so initialization should /
     /  be performed.    If ulFlag is 1 then the DLL is being freed so           /
     /  termination should be performed.                                         /

     switch (ulFlag) {
          case   :
```

Figure 18 (Part 2 of 4). INITTERM.C - _DLL_InitTerm *Function for SAMPLE03*

```
            /                                              /
            / The C run-time environment initialization function must be          /
            / called before any calls to C run-time functions that are not       /
            /  inlined.                                                                          /
            /                                              /

            if (_CRT_init() == -1)
                return   UL;
#ifndef   STATIC_LINK

            /                                              /
            / A DosExitList routine must be used to clean up if runtime calls  /
            / are required and the runtime is dynamically linked.                      /
            /                                              /

                if (rc = DosExitList( x     FF  |EXLST_ADD, cleanup))
                printf("DosExitList returned %lu\n", rc);
#endif
            if (rc = DosQueryModuleName(hModule, CCHMAXPATH, namebuf))
                printf("DosQueryModuleName returned %lu\n", rc);
            else
                printf("The name of this DLL is %s\n", namebuf);
            srand(17);
            nSize = (rand()%128)+32;
            printf("The array size for this process is %u\n", nSize);
            if ((pArray = malloc(nSize  sizeof(int))) == NULL) {
                printf("Could not allocate space for unsorted array.\n");
                return   UL;
            }
            for (i =  ; i < nSize; ++i)
                pArray[i] = rand();
            break;
```

Figure 18 (Part 3 of 4). INITTERM.C - _DLL_InitTerm *Function for SAMPLE03*

```
        case 1 :
#ifdef    STATIC_LINK
            printf("The array will now be freed.\n");
           free(pArray);
           _CRT_term();
#endif
           break;
       default   :
            printf("ulFlag = %lu\n", ulFlag);
           return   UL;
   }

   /  A non-zero value must be returned to indicate success.                    /

   return  1UL;
}
#ifndef   STATIC_LINK
static void cleanup(ULONG ulReason)
{
    if (!ulReason) {
        printf("The array will now be freed.\n");
       free(pArray);
   }
   DosExitList(EXLST_EXIT,  cleanup);
   return  ;
}
#endif
```

Figure 18 (Part 4 of 4). INITTERM.C - _DLL_InitTerm *Function for SAMPLE03*

The SAMPLE03 sample program is described in more detail in “Sample Program to Build a DLL” on page 207.

Creating Resource DLLs

Resource DLLs contain application resources that your program uses, such as menus, bitmaps, and dialog templates. You can define these resources in a .RC file using OS/2 APIs, or with the Toolkit Icon Editor and Dialog Editor. Use the Toolkit Resource Compiler to build the resources into a DLL, which is then called by your executable program at run time.

The benefits of using a resource DLL instead of binding the resources directly into your executable file include easier maintenance and less duplication of resources. You may even be able to use a common resource DLL for multiple applications.

To create a resource DLL:

1. Create an empty source file. A resource DLL can contain only resources.
2. Create a .DEF file. The only statement required in this file is LIBRARY to specify that a DLL is to be built.
3. Create a .RC file that defines your resources. See the Toolkit documentation for more information on creating and defining resources.
4. Compile the source file using /C+ to specify compile only. For example:

   ```
   icc /C+ empty.c
   ```

   Do not specify the /Ge- option. Specifying /Ge- causes the DLL initialization and termination code to be included in the object module, and the resource DLL cannot contain code.
5. Link the resulting object module, using your .DEF file, to create an empty DLL:

   ```
   LINK386  empty.obj,resdll.dll,,,mydef.def
   ```

6. Compile your .RC file with the Resource Compiler to create a .RES file. For example:

   ```
   RC /r myres.rc
   ```

7. Use the Resource Compiler again to add the resources to the DLL. For example:

   ```
   RC myres.res resdll.dll
   ```

Your application can use OS/2 APIs to load the resource DLL and access the resources it contains. Like other DLLs, resource DLLs must be in a directory specified in your LIBPATH environment variable.

For more information on resources and the Resource Compiler, see the Toolkit *Programming Reference* and *Tools Reference*.

Creating Your Own Runtime Library DLLs

If you are shipping your application to other users, you can use one of two methods to make the C/C++ Tools runtime library functions available to the users of your application:

1. Statically bind every module to the library (.LIB) files.

   This method increases the size of your modules and also slows the performance because the library environment has to be initialized for each module. Having multiple library environments also makes signal handling, file I/O, and other operations more complicated.

2. Create your own runtime DLLs.

   This method provides one common runtime environment for your entire application. It also lets you apply changes to the runtime library without relinking your application, meaning that if the C/C++ Tools DLLs change, you need only rebuild your DLL.

Creating Runtime Library DLLs

To create your own runtime library, follow these steps:

1. Copy and rename the appropriate C/C++ Tools .DEF file for the program you are creating. For example, for a multithread program, copy DDE4MBS.DEF to myrtdll.def. You must also change the DLL name on the LIBRARY line of the .DEF file. The .DEF files are installed in the LIB subdirectory under the main C/C++ Tools installation directory.

2. Remove any functions you do not use directly or indirectly from the .DEF file, including the STUB line. Do not delete anything with the comment next to it; variables and functions indicated by this comments are always required because they are called by startup functions.

3. Create a source file for your DLL, for example, myrtdll.c. If you are creating a runtime library that contains only C/C++ Tools functions, create an empty source file. If you are adding your own functions to the library, put the code for them in this file.

4. Compile and link your DLL files. Use the /Ge- option to create a DLL, and the appropriate option for the type of DLL you are building (single-thread or multithread). For example, to create a multithread DLL, use the command:

   ```
   icc /Ge- /Gm+ myrtdll.c myrtdll.def
   ```

5. Use the IMPLIB utility from the Toolkit to create an import library for your DLL, as described in “Using Your DLL” on page 205. For example:

   ```
   IMPLIB /NOI myrtdlli.lib myrtdll.def
   ```

6. Use the WorkFrame/2 LIB utility to add the object modules that contain the initialization and termination functions to your import library. These objects are needed by all executable modules and DLLs, are contained in DDE4MBSO.LIB for multithread programs and DDE4SBSO.LIB for single-thread programs. See the WorkFrame/2 online documentation for information on how to use LIB.

   Note: If you do not use the WorkFrame/2 LIB utility, you must ensure that all objects that access your runtime DLL are statically linked to the appropriate object library. The compile and link commands are described in the next step.

7. Compile your executable modules and other DLLs with the /Gn+ option to exclude the default library information. For example:

```
icc /C /Gn+ /Ge+ myprog.c
icc /C /Gn+ /Ge- mydll.c
```

   When you link your objects, specify your own import library. If you are using or plan to use OS/2 APIs, specify OS2386.LIB also. For example:

```
LINK386 myprog.obj,,, myrtdlli.lib OS2386.LIB
LINK386 mydll.obj,,, myrtdlli.lib OS2386.LIB
```

   To compile and link in one step, use the commands:

```
icc /Gn+ /Ge+ myprog.c myrtdlli.lib OS2386.LIB
icc /Gn+ /Ge- mydll.c myrtdlli.lib OS2386.LIB
```

   Note: If you did not use the WorkFrame/2 LIB utility to add the initialization and termination objects to your import library, when you link your modules, specify:

   a. DDE4SBSO.LIB or DDE4MBSO.LIB
   b. Your import library
   c. OS2386.LIB (to allow you to use OS/2 APIs)
   d. The linker option /NOD.

   For example:

```
LINK386 /NOD myprog.obj,,,DDE4SBSO.LIB myrtdlli.lib OS2386.LIB;
LINK386 /NOD mydll.obj,,,DDE4SBSO.LIB myrtdlli.lib OS2386.LIB;
```

   The /NOD option tells the linker to disregard the default libraries specified in the object files and use only the libraries given on the command line. If you are using icc to invoke the linker for you, the commands would be:

```
icc /B"/NOD" myprog.c DDE4SBSO.LIB myrtdlli.lib OS2386.LIB
icc /Ge- /B"/NOD" mydll.c DDE4SBSO.LIB myrtdlli.lib OS2386.LIB
```

   The linker then links the objects from the object library directly into your executable module or DLL.

Example of Creating a Runtime Library

In the sample program SAMPLE03, the program MAIN 3.C calls printf and srand from the C/C++ Tools runtime DLLs, and uses other variables and functions from SAMPLE 3.DLL. Because SAMPLE 3.DLL also uses printf and is statically linked to the runtime libraries, the code for the C/C++ Tools runtime functions it uses is linked into SAMPLE 3.DLL.

If these functions are included in SAMPLE 3.DLL, all external references from MAIN 3.C can be resolved by dynamically linking to this DLL. As a result, MAIN 3.EXE will be smaller.

Note: The process described here is only possible when the user DLL links statically to the C/C++ Tools runtime library.

Rebuild SAMPLE 3.DLL to include printf and srand as exports by following these steps:

1. Add _printfieee and srand to SAMPLE 3.DEF under the EXPORTS keyword.

   Note: When the language level is /Se, printf is mapped to _printfieee to support the IEEE extensions (infinity and NaN).

2. Use DDE4SBS.DEF to find what functions and variables must be exported, and add them to SAMPLE 3.DEF as EXPORTS.
3. Relink SAMPLE 3.DLL as described in "Compiling and Linking Your DLL" on page 203.

After your changes, SAMPLE 3.DEF should look like Figure 19. The example shown in this figure is actually the file SAMPLE3R.DEF, which is provided with the SAMPLE03 program.

```
LIBRARY SAMPLE 3 INITINSTANCE TERMINSTANCE
PROTMODE
DATA MULTIPLE NONSHARED READWRITE LOADONCALL
CODE LOADONCALL
EXPORTS
     nSize        ; array size
     pArray       ; pointer to base of array of ints
     nSwaps       ; number of swaps required to sort the array
     nCompares ; number of comparisons required to sort the array
     list         ; array listing function
     bubble       ; bubble sort function
;    selection    selection sort function
     insertion ; insertion sort function
                  ; CRT symbols required by EXE
    _printfieee
    srand
    _critlib_except                          ;
    _DosSelToFlat                            ;
    _DosFlatToSel                            ;
    _environ                                 ;
    _CRT_init                                ;
    __ctordtorInit                           ;
    _EXE_Exception                           ;
    _Exception                               ;
    _PrintErrMsg                             ;
    _exception_procinit                      ;
    _exception_dllinit                       ;
    _matherr                                 ;
```

Figure 19 (Part 1 of 2). SAMPLE3R.DEF - Definition File to Export C Runtime Functions

```
_terminate          ;
__ctordtorTerm      ;
exit                ;
free                ;
malloc              ;
strdup              ;
strpbrk             ;
```

Figure 19 (Part 2 of 2). SAMPLE3R.DEF - Definition File to Export C Runtime Functions

Once you have relinked SAMPLE 3.DLL, re-create MAIN 3.EXE so the calls to the C/C++ Tools runtime functions are resolved by dynamically linking to SAMPLE 3.DLL. A make file, MAKE 3R, is provided to do this for you.

Note: You must have the Toolkit installed to use the make file.

To re-create MAIN 3.EXE, at the prompt in the SAMPLES\SAMPLE03 directory, type:

```
nmake all /f MAKE 3R
```

To recompile and relink MAIN 3.EXE yourself:

1. Use the IMPLIB utility to create an import library from SAMPLE 3.DEF, using the command:

   ```
   IMPLIB SAMPLE 3.LIB SAMPLE 3.DEF
   ```

2. Compile and link MAIN 3.EXE with the command:

   ```
   icc /B"/NOE /NOD" MAIN 3.C DDE4SBSO.LIB SAMPLE 3.LIB OS2386.LIB
   ```

   Note: If you compiled with the option /Gn+, the linker option /NOD is not required, but you must recompile all the modules with this option.

   If MAIN 3.OBJ already exists, you can use the following command to create MAIN.EXE by simply relinking:

   ```
   LINK386 /NOI /NOE /NOD MAIN 3,,,DDE4SBSO SAMPLE 3 OS2386;
   ```

Creating Runtime Library DLLs

After you have performed these steps, copy SAMPLE 3.DLL to a directory listed in the LIBPATH variable in your CONFIG.SYS file. You can then use the command:

```
MAIN 3
```

to run the SAMPLE03 program.

Part 5. Advanced Topics

This part describes some of the advanced features of the C/C++ Tools compiler.

Advanced Topics

Chapter 13. Using Templates in C++ Programs

This chapter describes how the compiler generates template function bodies and how you should structure your program to use templates.

Note: It is important to note the distinction between the terms *function template* and *template function*:

A *function template* is a template used to generate template functions. A function template can be only a declaration or it can define the function.

A *template function* is a function declared or defined by a function template.

For a general description of templates, see the *Online Language Reference* or the *C++ Language Reference*.

Important: When you link C++ object files, you must use the icc command with the /Tdp option to invoke the linker. If you invoke the linker in a separate step (with the LINK386 command), the template functions may not resolve correctly.

Generating Template Function Definitions

When you use class and function templates in your program, the C/C++ Tools compiler generates function bodies automatically for all template functions that are:

1. Referenced in the source code
2. Defined by a function template that is visible to that source
3. Not explicitly defined by the user.

In each compilation unit where a template function that meets these criteria appears, the compiler generates a function body. At link time, all references to the function are resolved to a single generated body. Note that if you explicitly define a template function, all references are resolved to the explicit definition.

When you specify /Ft+ (which is the default), you can use a different method of template generation using *template-implementation files*, as described in “Using Template-Implementation Files” on page 229.

Generating Template Function Definitions

When you use this method, the compiler generates only one function body to be used for all compilation units.

Template functions with internal linkage are treated differently from those that are visible to other compilation units. A template function has internal linkage if it is either:

- Defined inline (meaning within a template class).
- Declared with the keyword inline.
- A non-member function declared with the keyword static.

If your template function has internal linkage, it is not visible outside of the compilation unit it resides in and must therefore be defined within that compilation unit. You can define it either by including the function template or by providing an explicit definition.

If the same template function is used in more than one compilation unit, the compiler generates a function body for each one. If the template function has internal linkage, each function body is used only in its own compilation unit and all definitions are kept at link time. If the template function does not have internal linkage, the compiler resolves the multiple definitions just before link time as follows:

1. If a compilation unit explicitly defines the function, all references are resolved to that definition. All other definitions are ignored. If more than one compilation unit explicitly defines the function, an error is generated.

2. If there is no explicit definition, the compiler uses one of the generated function definitions. All other definitions are ignored.

Note: The linker does not discard unused template function definitions when it creates your executable module. Repeating template declarations that define functions in multiple compilation units can make your executable modules very large. To avoid this problem, use the automatic template generation facility described in the following section, or structure your program so that the defining function templates are included in fewer compilation units.

Example of Generating Template Function Definitions

The class template, Stack, provides an example of template function generation. Stack implements a stack of items. The overloaded operators << and >> are used to push items on to the stack and pop items from the stack. Assume the declaration of the Stack class template is contained in the file stack.h:

```
template <class Item, int size> class Stack {
  public:
      int operator << (Item item);    // push operator
      int operator >> (Item& item); // pop operator
      Stack() { top =  ; }              // constructor defined inline
  private:
      Item stack[size];                 // stack of items
      int     top;                      // index to top of stack
};
```

Figure 20. Declaration of Stack in stack.h

In the template, the constructor function is defined inline. Assume the other functions are defined using separate function templates in the file stack.c:

```
template <class Item, int size>
    int Stack<Item,size>::operator << (Item item) {
        if (top >= size) return  ;
          stack[top++] = item;
         return  1;
    }
template <class Item, int size>
     int Stack<Item,size>::operator >> (Item& item)
    {
         if (top <=  ) return  ;
           item = stack[--top];
          return  1;
    }
```

Figure 21. Definition of operator Functions in stack.c

In this example, the constructor has internal linkage because it is defined inline in the class template declaration. In each compilation unit that uses an instance of the `Stack` class, the compiler generates the constructor function body. Each unit has its own copy of the constructor that it alone uses.

In each compilation unit that includes the file `stack.c`, for any instance of the `Stack` class in that unit the compiler generates definitions for the functions:

```
Stack<item,size>::operator<<(item)
Stack<item,size>::operator>>(item&)
```

For example, given the following source file:

```
#include "stack.h"
#include "stack.c"

 void Swap(int i&, Stack<int,2 >& s)
{
    int j;
     s >> j;
     s << i;
     i = j;
}
```

the compiler generates the functions `Stack<int,2 >::operator<<(int)` and `Stack<int,2 >::operator>>(int&)`.

Using the Compiler's Automatic Template Generation Facility

To avoid producing multiple definitions for your template functions, you can use the compiler's automatic template generation facility to generate the definition in one source file only. This is the recommended way to use templates with the C/C++ Tools compiler.

To use this facility, you declare or reference the template functions in your source, but do not define them. Instead, provide the definitions in a special file called a *template-implementation file*. The compiler uses this file to determine what instances of the template function must be created. It then creates an additional source file, called a *template-include file*, that contains the function definitions. No more than one definition is generated for each template function.

Using Template-Implementation Files

To create and use a template-implementation file:

Declare your template functions using class or function template declarations. If the function is a member of a template class, its declaration is part of the class template declaration. If the function is a nonmember function, you must declare the function using a function template. **Do not define the function**.

Place your class or function template declarations in a header file and include the file in your source code using the #include directive.

Create a template-implementation file for each header file that contains these template declarations. Give the file the same name as the header file but with the extension .c instead of .h. Place the template-implementation file in the same directory as the corresponding .h file.

Define all the functions declared in the header file in the template-implementation file. The definitions can be explicit function definitions, template definitions, or both. If you use explicit definitions, they are used instead of the definitions generated by the template.

Define any classes that are used in template arguments and that are required to generate the template function in the header file. If the class definitions require other header files, include them with the #include directive. The class definitions are then available in the template-implementation file when the function definition is compiled. Do not put the definitions of any classes used in template arguments in your source code.

Using Template-Implementation Files

In the Stack example, the file stack.c is a template-implementation file. To create a program using the Stack class template, you would include stack.h in any source files that use an instance of the class. The stack.c file does not need to be included in any source files, but must reside in the same directory as stack.h. Then given the source file:

```
#include "stack.h"

 void Swap(int i&, Stack<int,2 >& s)
{
    int j;
     s >> j;
     s << i;
     i = j;
}
```

the compiler automatically generates the functions Stack<int,2 >::operator<<(int) and Stack<int,2 >::operator>>(int&).

You can change the name of the template-implementation file or place it in a different directory using the #pragma implementation directive. This #pragma directive has the format:

```
#pragma implementation(string-literal)
```

where *string-literal* is the path name for the template-implementation file. If it is only a partial path name, it must be relative to the directory containing the header file.

For example, in the Stack class, to use the file stack.def as the template-implementation file instead of stack.c, add the line:

```
#pragma implementation("stack.def")
```

anywhere in the stack.h file. The compiler then looks for the template-implementation file stack.def in the same directory as stack.h.

Generating Template-Include Files

When it compiles your program, the compiler builds a template-include file for each header file that contains template functions that need to be defined. The compiler stores the template-include files in the TEMPINC subdirectory under the current directory. The compiler creates the TEMPINC directory if it does not already exist.

Before it invokes the linker, the compiler checks the contents of the TEMPINC subdirectory, compiles the template-include files, and generates the necessary template function definitions.

By default, the compiler places all template-include files in the TEMPINC subdirectory of the current directory. To redirect these files to another directory, use the /Ft*dir* compiler option, where *dir* is the directory to contain the template-include files. You can specify a fully-qualified path name or a path name relative to the current directory.

If you specify a different directory for your template-include files, ensure you specify it consistently for all compilations of your program, including the link step.

Note that after the compiler creates a template-include file, it may add information to the file as each compilation unit is compiled. However, the compiler never removes information from the file. If you remove function instantiations or reorganize your program so that the template-include files become obsolete, you may want to delete one or more of these files and recompile your program. In addition, if error messages are generated for a file in the TEMPINC directory, you must either correct the errors manually or delete the file and recompile. To regenerate all of the template-include files, delete the TEMPINC directory and recompile your program.

Contents of a Template-Include File

The following example shows the information you would find in a typical template-include file generated by the compiler:

```
/ 698421265 / #include "\swearsee\src\list.h"          1
/           / #include "\swearsee\src\list.c"         2
/ 698414 46 / #include "\swearsee\src\mytype.h"       3
/ 698414 46 / #include "\IBMCPP\INCLUDE\iostream.h"  4
#pragma define(List<MyType>)                           5
ostream& operator<<(ostream&,List<MyType>);           6
#pragma undeclared                                     7
int count(List<MyType>);                              8
```

1 The header file that corresponds to the template-include file. The number in comments at the start of each #include line (for this line / 698421265 /) is a time stamp for the included file. The compiler uses this number to determine if the template-include file is current or should be recompiled. A time stamp containing only zeroes ( ) as in line 2 means the compiler is to ignore the time stamp.

2 The template-implementation file that corresponds to the header file in line 1

3 Another header file that the compiler requires to compile the template-include file. All other header files that the compiler needs to compile the template-include file are inserted at this point.

4 Another header file required by the compiler. It is referenced in the function declaration in line 6 .

5 The compiler inserts #pragma define directives to force the definition of template classes. In this case, the class List<MyType> is to be defined and its member functions are to be generated.

6 The operator<< function is a nonmember function that matched a template declaration in the list.h file. The compiler inserts this declaration to force the generation of the function definition.

7 The #pragma undeclared directive is used only by the compiler and only in template-include files. It is used to separate functions that were instantiated using a declaration from functions that were instantiated using a call. All template functions that were explicitly declared in at least one compilation unit appear before this line. All template functions that were called, but never declared, appear after this line.

8 count is an example of a template function that was called but not declared. The template declaration of the function must have been contained in list.h, but the instance count(List<MyType>) was never declared.

Note: Although you can edit the template-include files, it is not normally necessary or advisable to do so.

Structuring Your Program for Templates Manually

If you do not want to use the template-implementation file method of generating template function definitions, you can structure your program in such a way that you define template functions directly in your compilation units. If you structure your program manually, you do not have to reference any compiler-generated files, but if you change the body of the function template you may need to recompile many source files. In addition, compile and link time may be longer and the object file produced can become quite large because of multiple definitions.

Note: It is recommended that you use the compiler's automatic template generation facility.

Structuring Your Program for Templates Manually

There are two ways you can structure your program to directly define template functions:

1. Include the function template definition in all compilation units that reference the corresponding template functions.
2. Include the declaration of the function template in all files that reference the corresponding template functions, but group the function definitions into a single compilation unit. (Note that this is essentially what the compiler does for you automatically when you use template-implementation files.) Use #pragma define directives to force the compiler to generate the necessary definitions for all template functions and classes used in other compilation units.

For example, to use the first method with the Stack template, include both stack.h and stack.c in all compilation units that use an instance of the Stack class. The compiler then generates definitions for each template function. Each template function may be defined multiple times, increasing the size of the object file.

To use the second method, include stack.h in all compilation units that use an instance of the Stack class, but include stack.c in only one of the files. Alternatively, if you know what instances of the Stack class are being used in your program, you can define all of the instances in a single compilation unit. For example:

```
#include  "stack.h"
#include  "stack.c"
#include "myclass.h"    // Definition of "myClass" class
#pragma  define(Stack<int,2 >)
#pragma  define(Stack<myClass,1  >)
```

The #pragma define directive forces the definition of two instances of the Stack class without creating any object of the class. Because these instances reference the member functions of that class, template function definitions are generated for those functions. See the *Online Language Reference* for more information about the #pragma directive.

When you use these methods, you may also need to specify the /Ft- option to ensure that the compiler does not look for template-implementation files to resolve the template functions. Note that specifying /Ft- does not affect the compiler's generation of template functions as described in "Generating Template Function Definitions" on page 225.

Using Static Data Members in Templates

Partial support for using static data members within templates has been provided in this version of C/C++ Tools. You can use templates to define static data members, but you must observe the following restrictions:

- You cannot combine template definitions of static data members with explicit definitions. If you try to use a static member template in one compilation unit and an explicit definition in another, the linker generates an error about multiple definitions.
- Static data members defined by templates are not visible as dictionary entries in libraries. If your program references a static member defined in a library object, but does not reference any other external symbols in that object, the linker will not extract the object from the library.
- If you export a class with data members from a DLL, export the data members as well, regardless of their access specifiers (private, protected, or public).

When the compiler finds a template that defines static data members, it always generates a warning message (EDC3402). Note that if this message is generated from a class library header file that you have included in your code, you can ignore it because the data member has been used according to the restrictions.

Using Static Data Members in Templates

Chapter 14. Calling Conventions

This chapter describes the calling conventions used by the C/C++ Tools compiler for both C and C++:

- _Optlink
- _System
- _Pascal and _Far32 _Pascal
- 32/16-bit conventions:
  - _Far16 _Cdecl
  - _Far16 _Pascal
  - _Far16 _Fastcall

The _Optlink convention is specific to the C/C++ Tools compiler and is the fastest method of calling C or C++ functions or assembler routines, but it is not standard for all OS/2 applications. The _Optlink calling convention is described in more detail in "_Optlink Calling Convention" on page 238.

The _System calling convention, while slower, is standard for all OS/2 applications and is used for calling OS/2 APIs. See "_System Calling Convention" on page 264 for a description of the _System calling convention.

The _Pascal and _Far32 _Pascal conventions are used to develop virtual device drivers. The _Far32 _Pascal convention can only be used for applications written in C that run at ring zero (compiled with the /Gr+ option). More information about the _Pascal and _Far32 _Pascal conventions can be found in "_Pascal and _Far32 _Pascal Calling Conventions" on page 272.

Note: You cannot call a function using a different calling convention than the one with which it is compiled. For example, if a function is compiled with _System linkage, you cannot later call it specifying _Optlink linkage.

The different methods of calling 16-bit code from the C/C++ Tools compiler and the 16-bit calling conventions are discussed in Chapter 16, "Calling Between 32-Bit and 16-Bit Code" on page 287.

_Optlink Calling Convention

You can specify the calling convention for all functions within a program using the /Mp or /Ms compiler option. You can also use linkage keywords to specify the calling convention for individual functions. In C programs, you can also use #pragma linkage. Linkage keywords and the #pragma directive take precedence over the compiler option, if both are specified.

| **Note:** C++ member functions always use the _Optlink calling
| convention. You cannot change the convention for member functions.

See "Setting the Calling Convention" on page 62 for more details on setting the calling convention. For more information on compiler options, see "Code Generation Options" on page 111. For information about linkage keywords and #pragma linkage, see the *Online Language Reference*.

_Optlink Calling Convention

This is the default calling convention. It is an alternative to the _System convention that is normally used for calls to the operating system. It provides a faster call than the _System convention, while ensuring conformance to the ANSI and SAA language standards.

| You can explicitly give a function the _Optlink convention with the
| _Optlink keyword, or for C files only, the #pragma linkage directive.

Features of _Optlink

- Parameters are pushed from right to left onto the stack to allow for varying length parameter lists without having to use hidden parameters.
- The caller cleans up the parameters.
- The general-purpose registers EBP, EBX, EDI, and ESI are preserved across the call.
- The general-purpose registers EAX, ECX, and EDX are not preserved across the call.
- Floating-point registers are not preserved across the call.

The three conforming parameters that are lexically leftmost (conforming parameters are 1, 2, and 4-byte signed and unsigned integers, enumerations, and all pointer types) are passed in the three unpreserved general-purpose registers.

Up to four floating-point parameters (the lexically first four) are passed in extended-precision format (80-bit) in the floating-point register stack.

All other parameters are passed on the 80386 stack.

Space for the parameters in registers is allocated on the stack, but the parameters are not copied into that space.

Conforming return values are returned in EAX.

Floating-point return values are returned in extended-precision format in the topmost register of the floating-point stack.

Complex floating-point return values are returned in extended-precision format in the topmost two registers of the floating-point stack.

When you call external functions, the floating-point register stack contains only valid parameter registers on entry and valid return values on exit. (When you call functions in the current compilation unit that do not call any other functions, this state may not be true.)

Calls with aggregates returned by value pass a hidden first parameter that is the address of a storage area determined by the caller. This area becomes the returned aggregate. The address of this aggregate is returned in EAX.

The direction flag must be clear upon entry to functions and clear on exit from functions. The state of the other flags is ignored on entry to a function and undefined on exit.

The compiler will not change the contents of the floating-point control register. If you want to change the control register contents for a particular operation, save the contents before making the changes and restore them after the operation.

Tips for Using _Optlink

To obtain the best performance when using the _Optlink convention, follow these tips:

- Prototype all function declarations for better performance. The C++ language requires all functions to have prototypes.

  Note: All calls and functions must be prototyped consistently; that is, functions declared more than once must have identical prototypes. If prototyping is not consistent, the results will be undefined.

- Place the conforming and floating-point parameters that are most heavily used lexically leftmost in the parameter list so they will be considered for registers first. If they are adjacent to each other, the preparation of the parameter list will be faster.
- If you have a parameter that is only used near the end of a function, put it at or near the end of the parameter list. If all of your parameters are only used near the end of functions, consider using _System linkage.
- If you are passing structures by value, put them at the end of the parameter list.
- Avoid using variable arguments in nonprototype functions. This practice results in undefined behavior under the ANSI C standard.
- If you have a variable-length argument list, consider using _System linkage. It is faster in this situation.
- Compile with optimization on by specifying /O+.

For additional tips on how to improve the performance of your program, see Chapter 10, "Optimizing Your Program" on page 165.

General-Purpose Register Implications

Parameters

EAX, EDX, and ECX are used for the lexically first three conforming parameters with EAX containing the first parameter, EDX the second, and ECX the third. Four bytes of stack storage are allocated for each register parameter that is present, but the parameters exist only in the registers at the time of the call.

If there is no prototype or an incomplete prototype for the function called, an eyecatcher is placed after the call instruction to tell the callee how the register parameters correspond to the stack storage mapped for them. Fully prototyped code never needs eyecatchers.

Eyecatchers

An eyecatcher is a recognizable sequence of bytes that tells unprototyped code which parameters are passed in which registers. Eyecatchers apply only to code without prototype statements.

The eyecatcher instruction is placed after the call instruction for a nonprototype function. The choice of instruction for the eyecatcher relies on the fact that the TEST instruction does not modify the referenced register, meaning that the return register of the call instruction is not modified by the eyecatcher instruction. The absence of an eyecatcher in unprototyped code implies that there are no parameters in registers.[3]

The eyecatcher has the format:

```
TEST EAX, immed32
```

Note that the short-form binary encoding (A9) of TEST EAX must be used for the eyecatcher instruction.

3 Note that this eyecatcher scheme does not allow the use of execute-only code segments.

Eyecatchers

The 32-bit immediate operand is interpreted as a succession of 2-bit fields, each of which describes a register parameter or a 4-byte slot of stack memory. Because only one 32-bit read of the eyecatcher is made, only 24 bits of the immediate operand are loaded. The actual number of parameters that can be considered for registers is restricted to 12.

Because of byte reversal, the bits that are loaded are the low-order 24 bits of the 32-bit immediate operand. The highest-order 2-bit field of the 24 bits analyzed corresponds to the lexically first parameter, while subsequent parameters correspond to the subsequent lower-order 2-bit fields. The meaning of the values of the fields is as follows:

Value	Meaning
00	A 4-byte slot of the parameter list in its 4-byte slot on the stack and not in any register. It indicates that there are no parameters remaining to be put into registers, or that there are parameters that could be put into registers but there are no registers remaining. It also indicates the end of the eyecatcher.
01	The corresponding parameter is in a general-purpose register. The leftmost field of this value has its parameter in EAX, the second leftmost (if it exists) in EDX, and the third (if it exists) in ECX.
10	The corresponding parameter is in a floating-point register and has 8 bytes of stack reserved for it (that is, it is a double). ST(0), ST(1), ST(2), and ST(3) contain the lexically-first four floating-point parameters (fewer registers are used if there are fewer floating-point parameters). ST(0) contains the lexically first (leftmost 2-bit field of type 10 or 11) parameter, ST(1) the lexically second parameter, and so on.
11	The corresponding parameter is in a floating-point register and has 16 bytes of stack reserved for it (that is, it is a long double).

Examples of Passing Parameters

The examples on the following pages are included for purposes of illustration and clarity only and have not been optimized. These examples assume that you are familiar with programming in assembler. Note that, in each example, the stack grows toward the bottom of the page, and ESP always points to the top of the stack.

Passing Conforming Parameters to a Prototyped Routine

The following example shows the code sequences and a picture of the stack for a call to the function foo:

```
long foo(char p1, short p2, long p3, long p4);

short  x;
long  y;

y = foo('A', x, y+x, y);
```

Caller's code surrounding call:

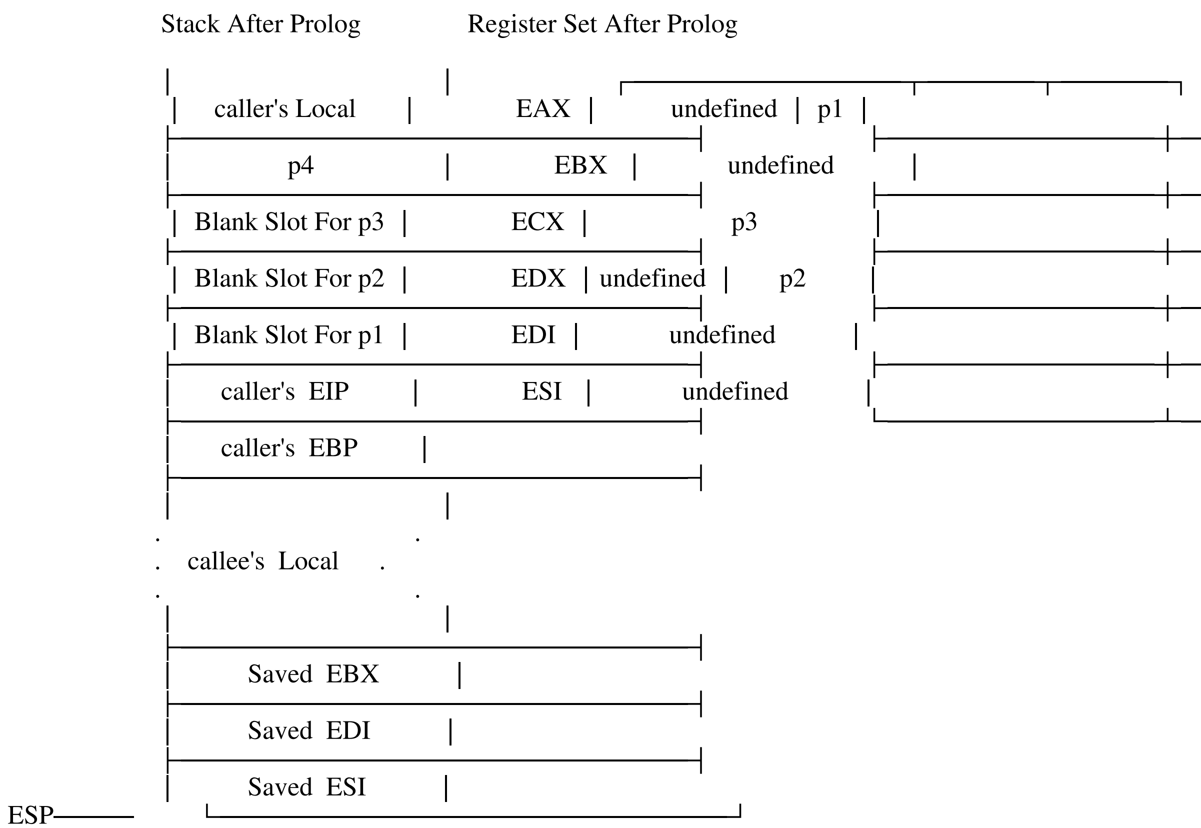
```
PUSH     y            ; Push p4 onto the 8 386 stack
SUB      ESP, 12      ; Allocate stack space for
                      ;    register  parameters
MOV      AL, 'A'      ; Put p1 into AL
MOV      DX,   x      ; Put p2 into DX
MOVSX    ECX, DX      ; Sign-extend x to long
ADD      ECX, y       ; Calculate p3 and put it into ECX
CALL     FOO          ; Make call
```

Examples Using _Optlink

```
Stack Just After Call            Register Set Just After Call

| caller's Local       |    EAX  |  undefined  | p1 |
|----------------------|
| p4                   |    EBX  |  caller's EBX     |
|----------------------|
| Blank Slot For p3    |    ECX  |  p3               |
|----------------------|
| Blank Slot For p2    |    EDX  | undefined |  p2   |
|----------------------|
| Blank Slot For p1    |    EDI  |  caller's EDI     |
|----------------------|
| caller's EIP         |    ESI  |  caller's ESI     |
ESP-----
```

Callee's prolog code:

```
PUSH    EBP                        ; Save caller's EBP
MOV     EBP, ESP                   ; Set up callee's EBP
SUB     ESP, callee's local size ; Allocate callee's Local
PUSH    EBX                        ; Save preserved registers -
PUSH    EDI                        ;   will optimize to save
PUSH    ESI                        ;   only registers callee uses
```



Note: The term *undefined* in registers EBX, EDI and ESI refers to the fact that they can be safely overwritten by the code in foo.

Callee's epilog code:

```
MOV    EAX, RetVal ; Put return value in EAX
POP    ESI         ; Restore preserved registers
POP    EDI
POP    EBX
MOV    ESP, EBP    ; Deallocate callee's local
POP    EBP         ; Restore caller's EBP
RET                ; Return to caller
```

Examples Using _Optlink

Stack After Epilog		Register Set After Epilog
caller's Local	EAX	Return Value
p4	EBX	caller's EBX
Blank Slot For p3	ECX	undefined
Blank Slot For p2	EDX	undefined
ESP → Blank Slot For p1	EDI	caller's EDI
	ESI	caller's ESI

Caller's code just after call:

```
ADD     ESP, 16     ; Remove parameters from stack
MOV     y,   EAX    ; Use return value.
```

Stack After Cleanup		Register Set After Cleanup
ESP → caller's Local	EAX	Return Value
	EBX	caller's EBX
	ECX	undefined
	EDX	undefined
	EDI	caller's EDI
	ESI	caller's ESI

Passing Conforming Parameters to an Unprototyped Routine

This example differs from the previous one by providing:

- An eyecatcher after the call to foo in the caller's code
- The code necessary to perform the default widening rules required by ANSI
- The instruction to clean up the parameters from the stack.

If foo were an ellipsis routine with fewer than three conforming parameters in the invariant portion of its parameter list, it would also include the code to interpret the eyecatchers in its prolog.

```
y = foo('A', x, y+x, y);
```

Caller's code surrounding call:

```
PUSH    y           ; Push p4 onto the 8 386 stack
SUB     ESP, 12     ; Allocate stack space for register parameters
MOV     EAX, 41h    ; Put p1 into EAX (41 hex = A ASCII)
MOVSX   EDX, x      ; Put p2 into EDX
MOV     ECX, y      ; Load y to calculate p3
ADD     ECX, x      ; Calculate p3 and put it into ECX
CALL    FOO         ; Make call
TEST    EAX, 54 h   ; Eyecatcher indicating 3 general-purpose
                    ;   register parameters (see page 241)
ADD     ESP, 16     ; Clean up parameters after return
```

Passing Floating-Point Parameters to a Prototyped Routine

The following example shows code sequences, 80386 stack layouts, and floating-point register stack states for a call to the routine fred. For simplicity, the general-purpose registers are not shown.

```
double fred(float p1, double p2, long double p3, float p4, double p5);

double a, b, c;
float  d, e;

a = b + fred(a, d, (long double)(a + c), e, c);
```

Caller's code up until call:

```
PUSH    2ND DWORD OF c          ; Push upper 4 bytes of c onto stack
PUSH    1ST DWORD OF c          ; Push lower 4 bytes of c onto stack
FLD     DWORD_PTR e             ; Load e into 8 387, promotion
                                ;   requires no conversion code
FLD     QWORD_PTR a             ; Load a to calculate p3
FADD    ST( ), QWORD_PTR c      ; Calculate p3, result is long double
                                ;   from nature of 8 387 hardware
FLD     QWORD_PTR d             ; Load d, no conversion necessary
FLD     QWORD_PTR a             ; Load a, demotion requires conversion
FSTP    DWORD_PTR [EBP - T1]    ; Store to a temp (T1) to convert to float
FLD     DWORD_PTR [EBP - T1]    ; Load converted value from temp (T1)
SUB     ESP, 32                 ; Allocate the stack space for
                                ;   parameter  list
CALL    FRED                    ; Make call
```

Stack Just After Call	8 387 Register Set Just After Call	
caller's Local	ST(7)	Empty
Upper Dword of p5	ST(6)	Empty
Lower Dword of p5	ST(5)	Empty
Blank Dword for p4	ST(4)	Empty
Four	ST(3)	p4 (e)
Blank	ST(2)	p3 (a + c)
Dwords	ST(1)	p2 (d)
for p3	ST( )	p1 (a)
Two Blank		
Dwords for p2		
Blank Dword for p1		
caller's EIP		
ESP——		

Callee's prolog code:

```
PUSH    EBP                          ; Save caller's EBP
MOV     EBP, ESP                     ; Set up callee's EBP
SUB     ESP, callee's local size ; Allocate callee's Local
PUSH    EBX                          ; Save preserved registers -
PUSH    EDI                          ;   will optimize to save
PUSH    ESI                          ;   only registers callee uses
```

Examples Using _Optlink

```
Stack After Prolog          8 387 Register Set After Prolog

|                      |
|    caller's  Local   |      ST(7)     |   Empty      |
|----------------------|
| Upper Dword of p5    |      ST(6)     |   Empty      |
|- - - - - - - - - - - |
| Lower Dword of p5    |      ST(5)     |   Empty      |
|----------------------|
| Blank Dword for p4   |      ST(4)     |   Empty      |
|----------------------|
|        Four          |      ST(3)     |   p4         |
|- - - - - - - - - - - |
|        Blank         |      ST(2)     |   p3         |
|- - - - - - - - - - - |
|        Dwords        |      ST(1)     |   p2         |
|- - - - - - - - - - - |
|        for  p3       |      ST( )     |   p1         |
|----------------------|
|      Two  Blank      |
|- - - - - - - - - - - |
|    Dwords for p2     |
|----------------------|
| Blank Dword for p1   |
|----------------------|
|     caller's  EIP    |
|----------------------|
|     caller's  EBP    |
|----------------------|
|                      |
.                      .
.   callee's  Local    .
.                      .
|                      |
|----------------------|
|      Saved  EBX      |
|----------------------|
|      Saved  EDI      |
|----------------------|
|      Saved  ESI      |
ESP-------------------------
```

Callee's epilog code:

```
FLD     RETVAL      ; Load return value onto floating-point stack
POP     ESI         ; Restore preserved registers
POP     EDI
POP     EBX
MOV     ESP, EBP    ; Deallocate callee's local
POP     EBP         ; Restore caller's EBP
RET                 ; Return to caller
```

Stack After Epilog

caller's Local
Upper Dword of p5
Lower Dword of p5
Blank Dword for p4
Four Blank Dwords for p3
Two Blank Dwords for p2
Blank Dword for p1

ESP → Blank Dword for p1

8 387 Register Set After Epilog

ST(7)	Empty
ST(6)	Empty
ST(5)	Empty
ST(4)	Empty
ST(3)	Empty
ST(2)	Empty
ST(1)	Empty
ST( )	Return Value

Caller's code just after call:

```
ADD     ESP, 4          ; Remove parameters from stack
FADD    QWORD_PTR b     ; Use return value
FSTP    QWORD_PTR a     ; Store expression to variable a
```

```
                 Stack After Cleanup          8 387 Register Set After Cleanup

                 |                     |
                 |  caller's Local  |  |    ST(7)   |      Empty          |
                 |                     |
ESP──────────                                       ST(6)   |      Empty
                                            ST(5)   |      Empty          |
                                            ST(4)   |      Empty          |
                                            ST(3)   |      Empty          |
                                            ST(2)   |      Empty          |
                                            ST(1)   |      Empty          |
                                            ST( )   |   Return  Value     |
```

Passing Floating-Point Parameters to an Unprototyped Routine

This example differs from the previous floating-point example by the presence of an eyecatcher after the call to fred in the caller's code and the code necessary to perform the default widening rules required by ANSI.

```
 double a, b, c;
float   d,  e;

 a = b + fred(a, d, (long double)(a + c), e, c);
```

Caller's code up until call:

```
PUSH    2ND DWORD OF c          ; Push upper 4 bytes of c onto stack
PUSH    1ST DWORD OF c          ; Push lower 4 bytes of c onto stack
FLD     DWORD_PTR e             ; Load e into 8 387, promotion
                                ;   requires no conversion code
FLD     QWORD_PTR a             ; Load a to calculate p3
FADD    ST( ), QWORD_PTR c      ; Calculate p3, result is long double
                                ;   from nature of 8 387 hardware
FLD     QWORD_PTR d             ; Load d, no conversion necessary
FLD     QWORD_PTR a             ; Load a, no conversion necessary
SUB     ESP, 4                  ; Allocate the stack space for
                                ;   parameter  list
CALL    FRED                    ; Make call
TEST    EAX,   ae    h          ; Eyecatcher maps the register parameters
ADD     ESP, 48                 ; Clean up parameters from stack
```

Passing and Returning Aggregates by Value to a Prototyped Routine

If an aggregate is passed by value, the following code sequences are produced for the caller and callee:

'C' Source:

```
struct s_tag {
          long   a;
          float  b;
          long   c;
          } x, y;
long   z;
double  q;
```

Examples Using _Optlink

```
/ Prototype /
struct s_tag bar(long lvar, struct s_tag aggr, float fvar);

:

/ Actual Call /
y = bar(z, x, q);

:

/ callee /
struct s_tag bar(long lvar, struct s_tag aggr, float fvar)
{
     struct s_tag temp;

     temp.a = lvar + aggr.a + 23;
     temp.b = fvar - aggr.b;
     temp.c = aggr.c

    return  temp;
}
```

Caller's code up until call:

```
FLD      QWORD_PTR q            ; Load lexically first floating-point
                                ;    parameter to be converted
FSTP     DWORD_PTR [EBP - T1] ; Convert to formal parameter type by
FLD      DWORD_PTR [EBP - T1] ; Storing and loading from a temp (T1)
SUB      ESP, 4                 ; Allocate space for the floating-point
                                ;    register  parameter
PUSH     x.c                    ; Push nonconforming parameters on
PUSH     x.b                    ;    stack
PUSH     x.a                    ;
```

```
MOV    EAX, Z        ; Load lexically first conforming
                     ;    parameter into EAX
SUB    ESP, 4        ; Allocate stack space for the first
                     ;    general-purpose register parameter.
PUSH   addr y        ; Push hidden first parameter (address of
                     ;    return  space)
CALL   BAR
```

Stack Just After Call	General-Purpose Registers Just After Call	
caller's Local	EAX	z
Blank Slot for q	EBX	caller's EBX
x.c	ECX	undefined
x.b	EDX	undefined
x.a	EDI	caller's EDI
Blank Slot for z	ESI	caller's ESI
Hidden Ret Val Addr		
caller's EIP		
ESP———		

8 387 Register Set Just After Call

ST(7)	Empty
ST(6)	Empty
ST(5)	Empty
ST(4)	Empty
ST(3)	Empty
ST(2)	Empty
ST(1)	Empty
ST( )	fvar [(float)q]

Examples Using _Optlink

Callee's prolog code:

```
PUSH    EBP         ; Save caller's EBP
MOV     EBP, ESP    ; Set up callee's EBP
SUB     ESP, 12     ; Allocate callee's Local
                    ;    = sizeof(struct s_tag)
PUSH    EBX         ; Save preserved registers -
PUSH    EDI         ;    will optimize to save
PUSH    ESI         ;    only registers callee uses
```

Stack After Prolog

Register Set After Prolog

```
|                        |
|   caller's  Local      |     EAX  |   lvar | (z)     |
|   Blank Slot for q     |     EBX  |   caller's EBX   |
|         x.c            |     ECX  |   undefined      |
|         x.b            |     EDX  |   undefined      |
|         x.a            |     EDI  |   caller's  EDI  |
|   Blank Slot for z     |     ESI  |   caller's ESI   |
|  Hidden Ret Val Addr   |
|    caller's EIP        |
|    caller's EBP        |
|                        |
.                        .
.    callee's  Local     .
.                        .
|                        |
|     Saved  EBX         |
|     Saved  EDI         |
|     Saved  ESI         |
ESP---------
```

The term *undefined* in registers ECX and EDX refers to the fact that they can be safely overwritten by the code in *bar*.

8 387 Register Set Just After Call

```
ST(7) |  Empty              |
ST(6) |  Empty              |
ST(5) |  Empty              |
ST(4) |  Empty              |
ST(3) |  Empty              |
ST(2) |  Empty              |
ST(1) |  Empty              |
ST( ) |  fvar  [(float)q]   |
```

Examples Using _Optlink

Callee's code:

```
 temp.a = lvar + aggr.a + 23;
 temp.b = fvar - aggr.b;
 temp.c = aggr.c

return  temp;

ADD     EAX,  23                 ;
ADD     EAX, [EBP + 16]          ; Calculate temp.a
MOV     [EBP - 12], EAX          ;

FSUB    DWORD_PTR [EBP + 2 ]     ; Calculate temp.b
FSTP    DWORD_PTR [EBP - 8]      ;

MOV     EAX, [EBP + 24]          ; Calculate temp.c
MOV     [EBP - 4], EAX           ;

MOV     EAX, [EBP + 8]           ; Load hidden parameter (address
                                 ;    of return value storage). Useful
                                 ;    both for setting return value
                                 ;    and for returning address in EAX.

MOV     EBX, [EBP - 12]          ; Return temp by copying its contents
MOV     [EAX], EBX               ;    to the return value storage
MOV     EBX, [EBP - 8]           ;    addressed by the hidden parameter.
MOV     [EAX + 4], EBX           ;    String move instructions would be
MOV     EBX, [EBP - 4]           ;    faster above a certain threshold
MOV     [EAX + 8], EBX           ;    size of returned aggregate.
```

```
POP     ESI               ; Begin Epilog by restoring
POP     EDI               ;    preserved  registers.
POP     EBX
MOV     ESP, EBP          ; Deallocate callee's local
POP     EBP               ; Restore caller's EBP
RET                       ; Return to caller
```

Stack After Epilog
caller's Local
Blank Slot for q
x.c
x.b
x.a
Blank Slot for z
Hidden Return Value Address ← ESP

General-Purpose Registers After Epilog

Register	Contents
EAX	Addr of Return Value
EBX	caller's EBX
ECX	undefined
EDX	undefined
EDI	caller's EDI
ESI	caller's ESI

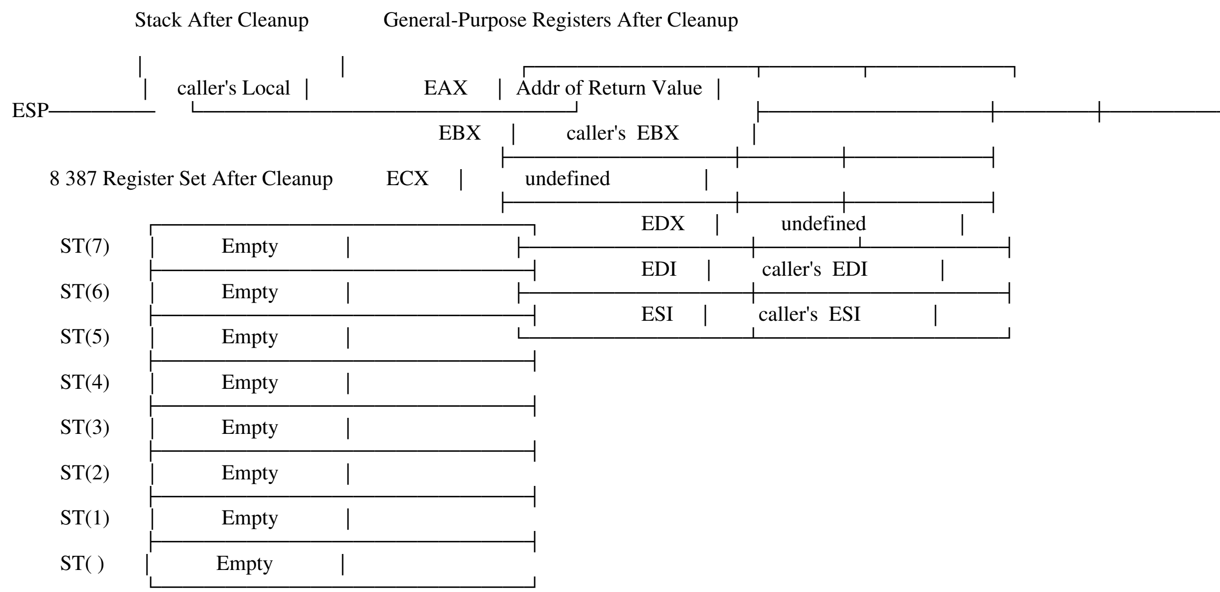
8 387 Register Set After Epilog

Register	Contents
ST(7)	Empty
ST(6)	Empty
ST(5)	Empty
ST(4)	Empty
ST(3)	Empty
ST(2)	Empty
ST(1)	Empty
ST( )	Empty

Examples Using _Optlink

Caller's code just after call:

```
ADD    ESP, 24        ; Remove parameters from stack
 ...                  ; Because address of y was given as the
                      ;   hidden parameter, the assignment of the
                      ;   return value has already been performed.
```



If a y.a = bar(x).b construct is used instead of the more common y = bar(x) construct, the address of the return value is available in EAX. In this case, the address of the return value (hidden parameter) would point to a temporary variable allocated by the compiler in the automatic storage of the caller.

Passing and Returning Aggregates by Value to an Unprototyped Routine

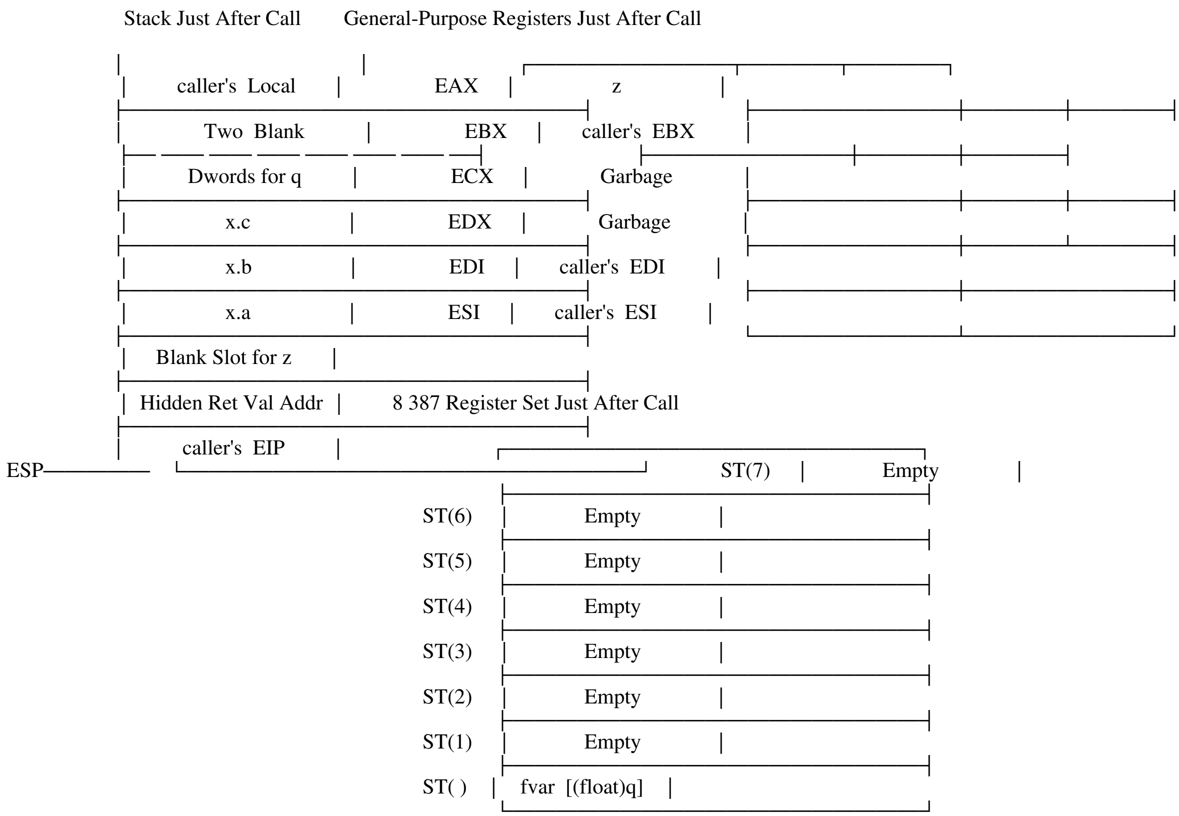
This example differs from the previous one by the presence of an eyecatcher after the call to bar in the caller's code and the code necessary to perform the default widening rules required by ANSI.

```
 struct s_tag {
                long    a;
                float  b;
                long    c;
                 } x, y;
long    z;
double  q;

 /  Actual Call  /
 y = bar(z, x, q);
  ...

 /  callee  /
 struct s_tag bar(long lvar, struct s_tag aggr, float fvar)
{
      struct s_tag temp;

      temp.a = lvar + aggr.a + 23;
      temp.b = fvar - aggr.b;
      temp.c = aggr.c

     return  temp;
}
```

Examples Using _Optlink

Caller's code up until call:

```
FLD     QWORD_PTR q     ; Load lexically first floating-point
                        ;   parameter to be converted
SUB     ESP, 8          ; Allocate space for the floating-point
                        ;   register parameter
PUSH    x.c             ; Push nonconforming parameters on
PUSH    x.b             ;   stack
PUSH    x.a             ;
MOV     EAX, z          ; Load lexically first
                        ;   conforming parameter
                        ;   into EAX
SUB     ESP, 4          ; Allocate stack space for the first
                        ;   general-purpose register parameter.
PUSH    addr y          ; Push hidden first parameter (address of
                        ;   return space)
CALL    BAR
TEST    EAX, 48h        ; Eyecatcher
ADD     ESP, 28         ; Clean up parameters
```


Stack Just After Call
General-Purpose Registers Just After Call
caller's Local
Two Blank
Dwords for q
x.c
x.b
x.a
Blank Slot for z
Hidden Ret Val Addr
caller's EIP
ESP
EAX
z
EBX
caller's EBX
ECX
Garbage
EDX
Garbage
EDI
caller's EDI
ESI
caller's ESI
8 387 Register Set Just After Call
ST(7)
Empty
ST(6)
Empty
ST(5)
Empty
ST(4)
Empty
ST(3)
Empty
ST(2)
Empty
ST(1)
Empty
ST( )
fvar [(float)q]

_System Calling Convention

To use this linkage convention, you must use the _System keyword in the declaration of the function, specify the /Ms option when you invoke the compiler, or for C files only, explicitly give a #pragma linkage directive.

Notes:

1. Because the C/C++ Tools library functions use the _Optlink convention, if you use the /Ms option, you must include all appropriate library header files to ensure the functions are called with the correct convention.
2. C++ member functions use the _Optlink convention. You cannot change the calling convention for member functions.

The following rules apply to the _System calling convention:

- All parameters are passed on the 80386 stack.
- The C parameter-passing convention is followed, where parameters are pushed onto the stack in right-to-left order.
- The calling function is responsible for removing parameters from the stack.
- All parameters are doubleword (4-byte) aligned.
- The size of the parameter list is passed in AL. If the parameter list is greater than 255 doublewords, the value contained in AL is the 8 least significant bits of the size. You can use the __parmdwords function (described in the *C Library Reference*) to access the value of AL that was passed to the function.
- All functions returning non-floating-point values pass a return value back to the caller in EAX. Functions returning floating-point values use the floating-point stack ST(0). Aggregate return values, such as structures, are passed as a hidden parameter on the stack, and EAX points to them on return.
- All functions preserve the general purpose registers of the caller, except for ECX, EDX, and EAX.
- Structures passed by value are actually copied onto the stack, not passed by reference.

The floating-point stack is defined to be empty upon entry to a called function, and has either a single item in ST(0) if there is a floating-point return, or is empty if there is not a floating-point return.

The direction flag must be clear upon entry to functions and clear on exit from functions. The state of the other flags is ignored on entry to a function, and undefined on exit.

The compiler will not change the contents of the floating-point control register. If you want to change the control register contents for a particular operation, save the contents before making the changes and restore them after the operation.

Examples Using the _System Convention

The following examples are included for purposes of illustration and clarity only and have not been optimized. The examples assume that you are familiar with programming in assembler. Note that, in the examples, the stack grows toward the bottom of the page, and ESP always points to the top of the stack.

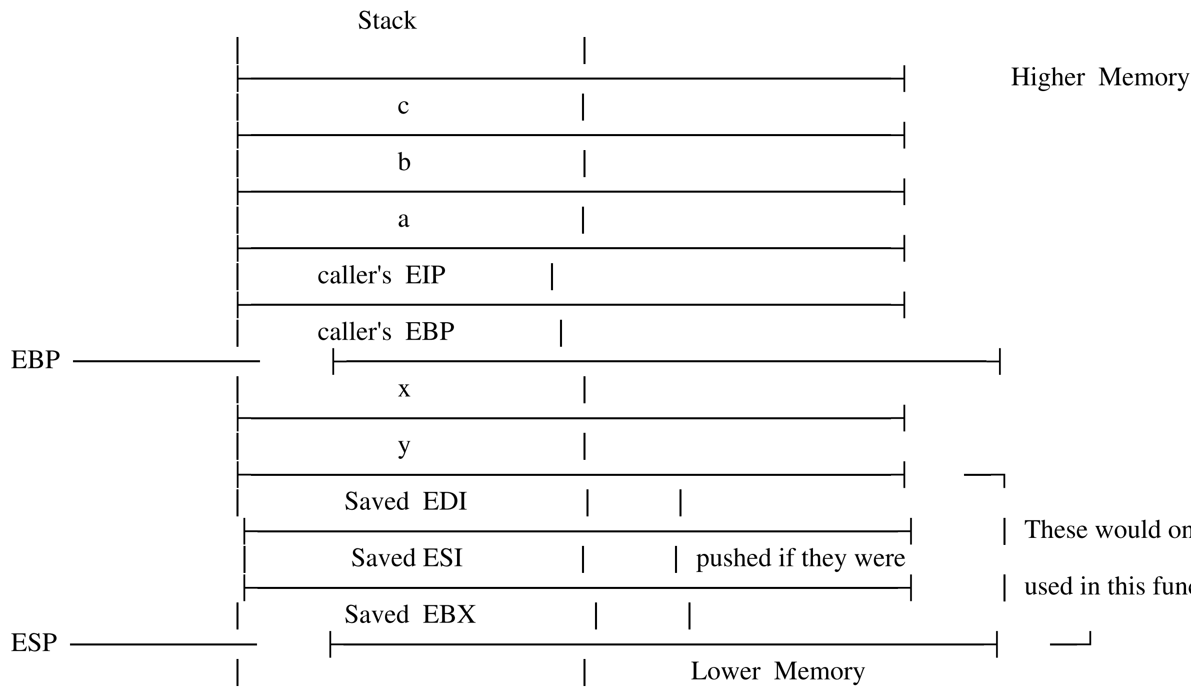
For the call

```
m = func(a,b,c);
```

a, b, and c are 32-bit integers and func has two local variables, x and y (both 32-bit integers).

Examples Using the _System Convention

The stack for the call to func would look like this:



The instructions used to create this activation record on the stack look like this on the calling side:

```
PUSH     c
PUSH     b
PUSH     c
MOV       AL, 3H
CALL     func
      .
      .
 ADD      ESP, 12     ; Cleaning up the parameters
      .
      .
MOV       m, EAX
      .
      .
```

For the callee, the code looks like this:

```
func PROC
   PUSH      EBP
    MOV       EBP, ESP          ;   Allocating 8 bytes of storage
    SUB      ESP, 8             ;    for two local variables.
    PUSH      EDI               ; These would only be
    PUSH      ESI               ;   pushed if they were used
    PUSH      EBX               ;   in this function.
   .
   .
    MOV       EAX, [EBP - 8]    ; Load y into EAX
    MOV       EBX, [EBP + 12]   ; Load b into EBX
   .
   .
    XOR       EAX, EAX          ; Zero the return value
    POP      EBX                ; Restore the saved registers
   POP       ESI
   POP       EDI
    LEAVE                       ; Equivalent to   MOV     ESP, EBP
                                ;                 POP     EBP
   RET
func ENDP
```

The saved register set is EBX, ESI, and EDI. The other registers (EAX, ECX, and EDX) can have their contents changed by a called routine.

Floating-point results are returned in ST(0) (the top of the floating-point register stack). If there is no numeric coprocessor installed in the system, the OS/2 operating system emulates the coprocessor.

Floating-point parameters are pushed on the 80386 stack.

Under some circumstances, the compiler will not use EBP to access automatic and parameter values, thus increasing the efficiency of the application. Whether it is used or not, EBP will not change across the call.

Examples Using the _System Convention

When passing structures as value parameters, the compiler generates code to copy the structure on to the 80386 stack. If the size of the structure is larger than an 80386 page size (4K), the compiler generates code to copy the structure backward. (That is, the last byte in the structure is the first to be copied.) This operation ensures that the OS/2 guard page method of stack growth will function properly in the presence of large structures being passed by value. Refer to "Controlling Stack Allocation and Stack Probes" on page 67 for more information on stack growth.

Structures are not returned on the stack. The caller pushes the address where the returned structure is to be placed as a lexically first hidden parameter. A function that returns a structure must be aware that all parameters are 4 bytes farther away from EBP than they would be if no structure return were involved. The address of the returned structure is returned in EAX.

In the most common case, where the return from a function is simply assigned to a variable, the compiler merely pushes the address of the variable as the hidden parameter.[4] For example:

```
struct test_tag
        {
        int  a;
        int  some_array[1  ];
        }  test_struct;

struct test_tag test_function(struct test_tag test_parm)
{
    test_parm.a = 42;
   return  test_parm;
}

int main(void)
{
    test_struct = test_function(test_struct);
   return  test_struct.a;
}
```

---

[4] Note that, if this function calls the __parmdwords function, the value of AL is stored in a temporary variable in its prolog. This is done to ensure that the value cannot change before the call to __parmdwords.

The code generated for this program would be:

```
test_function   PROC
    PUSH       ESI
    PUSH       EDI
     MOV        DWORD PTR [ESP+ cH],  2aH         ; test_parm.a
     MOV        EAX, [ESP+ 8H]                    ; Get the target of the return value
    MOV        EDI,  EAX                          ;  Value
    LEA        ESI,  [ESP+ cH]                    ;  test_parm
    MOV        ECX,   65H
    REP  MOVSD
    POP        EDI
    POP        ESI
    RET
test_function   ENDP

    PUBLIC    main
main   PROC
    PUSH       EBP
    MOV        EBP,  ESP
    PUSH       ESI
    PUSH       EDI

     SUB        ESP,  194H                        ; Adjust the stack pointer
    MOV        EDI,  ESP
     MOV        ESI, OFFSET FLAT: test_struct
    MOV        ECX,   65H
     REP MOVSD                                    ; Copy the parameter
    MOV        AL,   65H
     PUSH       OFFSET FLAT: test_struct          ; Push the address of the target
    CALL       test_function
    ADD        ESP,   198H

     MOV        EAX, DWORD PTR test_struct        ; Take care of the return
    POP        EDI                                ;    from  main
    POP        ESI
    LEAVE
    RET
main    ENDP
```

Examples Using the _System Convention

In a slightly different case, where only one field of the structure is used by the caller (as shown in the following example), the compiler allocates sufficient temporary storage in the caller's local storage area on the stack to contain a copy of the structure. The address of this temporary storage will be pushed as the target for the return value. Once the call is completed, the desired member of the structure can be accessed as an offset from EAX, as can be seen in the code generated for the example:

```
struct test_tag
        {
        int  a;
        int  some_array[1  ];
        }  test_struct;

struct test_tag test_function(struct test_tag test_parm)
{
     test_parm.a = 42;
    return  test_parm;
}

int main(void)
{
    return  test_function(test_struct).a;
}
```

The code generated for this example would be:

```
    PUBLIC    main
main    PROC
    PUSH      EBP
    MOV        EBP,  ESP
     SUB       ESP,  194H        ; Allocate space for compiler-generated
    PUSH      ESI                ;    temporary  variable
```

```
    PUSH      EDI
    SUB       ESP,  194H
    MOV       EDI,  ESP
    MOV       ESI, OFFSET FLAT: test_struct
    MOV       ECX,  65H
    REP  MOVSD
    LEA       EAX,  [ESP+ 19cH]
    PUSH      EAX
    MOV       AL,  65H
    CALL      test_function
    ADD       ESP,  198H
    MOV       EAX, [EAX]        ; Note the convenience of having the
    POP       EDI               ;   address of the returned structure
    POP       ESI               ;   in  EAX
    LEAVE
    RET
main    ENDP
```

_Pascal and _Far32 _Pascal Calling Conventions

The C/C++ Tools compiler provides both a _Pascal and a _Far32 _Pascal convention. The _Far32 _Pascal convention allows you to make calls between different code segments in code that runs at ring , and is only valid when the /Gr+ option is specified. The _Pascal conventions are most commonly used to create virtual device drivers, as described in Chapter 15, "Developing Virtual Device Drivers" on page 281.

Note: These _Pascal linkage conventions should not be confused with the 16-bit _Far16 _Pascal convention which is provided for 16-bit compatibility.

The _Pascal and _Far32 _Pascal conventions follow the same rules as the _System convention with these exceptions:

- Function names are converted to uppercase.
- Parameters are pushed in a left-to-right lexical order.
- The callee is responsible for cleaning up the parameters.
- Variable argument functions are not supported.
- The size of the parameter list is **not** passed in AL.

Important: The compiler does **not** convert 16-bit or 32-bit _Pascal function names to uppercase. The case of the function name in the call must match the case in the function prototype. Function names are however converted to uppercase in the object module to allow calls from assembler.

Examples Using the _Pascal Convention

The following examples are included for purposes of illustration and clarity only and have not been optimized. The examples assume that you are familiar with programming in assembler. Note that, in the examples, the stack grows toward the bottom of the page, and ESP always points to the top of the stack.

Examples Using the _Pascal Convention

For the call

```
m = func(a,b,c);
```

a, b, and c are 32-bit integers, and func has two local variables, x and y (both 32-bit integers).

The stack for the call to func would look like this:

```
                     Stack
              |                      |
              |---------------------------------------------|        Higher  Memory
              |          a           |
              |---------------------------------------------|
              |          b           |
              |---------------------------------------------|
              |          c           |
              |---------------------------------------------|
              |     caller's  EIP    |
              |---------------------------------------------|
              |     caller's  EBP    |
   EBP ------------      |---------------------------------------------|
              |          x           |
              |---------------------------------------------|
              |          y           |
              |---------------------------------------------|    --+
              |      Saved  EDI      |      |
              |---------------------------------------------|      | These would on
              |      Saved ESI       |      | pushed if they were
              |---------------------------------------------|      | used in this func
              |      Saved  EBX      |      |
   ESP ------------      |---------------------------------------------|    --+
              |                      |       Lower  Memory
```

Examples Using the _Pascal Convention

The instructions used to build this activation record on the stack look like this on the calling side:

```
PUSH    a
PUSH    b
PUSH    c
CALL    FUNC
   .
   .
   .
MOV     m, EAX
   .
   .
```

For the callee, the code looks like this:

```
FUNC PROC
    PUSH    EBP
    MOV     EBP, ESP          ;   Allocating 8 bytes of storage
    SUB     ESP, 8            ;    for two local variables.
    PUSH    EDI               ; These would only be
    PUSH    ESI               ;   pushed if they were used
    PUSH    EBX               ;   in this function.
    .
    .
    MOV     EAX, [EBP - 8]    ; Load y into EAX
    MOV     EBX, [EBP + 12]   ; Load b into EBX
    .
    .
```

```
    XOR      EAX, EAX           ; Zero the return value
    POP      EBX                ; Restore the saved registers
    POP      ESI
    POP      EDI
    LEAVE                       ; Equivalent to  MOV   ESP, EBP
                                ;                POP   EBP
    RET      CH
FUNC ENDP
```

Like the _System calling convention, the saved register set is EBX, ESI, and EDI. The other registers (EAX, ECX, and EDX) can have their contents changed by a called routine.

Floating-point results are returned in ST(0). If there is no numeric coprocessor installed in the system, the OS/2 operating system emulates the coprocessor. Floating-point parameters are pushed on the 80386 stack.

_Far32 _Pascal function pointers are returned with the offset in EAX and the segment in DX.

In some circumstances, the compiler will not use EBP to access automatic and parameter values, thus increasing the efficiency of the application. Whether it is used or not, EBP will not change across the call.

Structures are handled in the same way as they are under the _System calling convention. When passing structures as value parameters, the compiler generates code to copy the structure on to the 80386 stack. If the size of the structure is larger than an 80386 page size (4K), the compiler generates code to copy the structure backward. (That is, the last byte in the structure is the first to be copied.)

Structures are not returned on the stack. The caller pushes the address where the returned structure is to be placed as a lexically first hidden parameter. A function that returns a structure must be aware that all parameters are 4 bytes farther away from EBP than they would be if no structure return were involved. The address of the returned structure is returned in EAX.

Examples Using the _Pascal Convention

In the most common case, where the return from a function is simply assigned to a variable, the compiler merely pushes the address of the variable as the hidden parameter. For example:

```
struct test_tag {
          int  a;
          int  some_array[1  ];
          }  test_struct;

struct test_tag test_function(struct test_tag test_parm)
{
   test_parm.a = 42;
   return  test_parm;
}

int  main(void)
{
   test_struct = test_function(test_struct);
   return  test_struct.a;
}
```

The code generated for the above example would be:

```
TEST_FUNCTION   PROC
   PUSH      EBP
   MOV       EBP,  ESP
   PUSH      ESI
   PUSH      EDI
   MOV       DWORD PTR [ESP+ cH],  2aH     ; test_parm.a
   MOV       EAX, [EBP+ 8H]                ; Get the target of the return value
   MOV       EDI,  EAX                     ;  Value
   LEA       ESI,  [EBP+ cH]               ;  test_parm
```

```
   MOV        ECX,  65H
   REP  MOVSD
   POP        EDI
   POP        ESI
   LEAVE
   RET        198H
TEST_FUNCTION   ENDP

   PUBLIC    main
main   PROC
   PUSH       EBP
   MOV        EBP,  ESP
   PUSH       ESI
   PUSH       EDI

   SUB        ESP,  194H                       ; Adjust the stack pointer
   MOV        EDI,  ESP
   MOV        ESI, OFFSET FLAT: test_struct
   MOV        ECX,  65H
   REP MOVSD                                   ; Copy the parameter
   PUSH       OFFSET FLAT: test_struct       ; Push the address of the target
   CALL       TEST_FUNCTION

   MOV        EAX, DWORD PTR test_struct    ; Take care of the return
   POP        EDI                              ;    from  main
   POP        ESI
   LEAVE
   RET
main    ENDP
```

Examples Using the _Pascal Convention

In a slightly different case, where only one field of the structure is used by the caller (as shown in the following example), the compiler allocates sufficient temporary storage in the caller's local storage area on the stack to contain a copy of the structure. The address of this temporary storage will be pushed as the target for the return value. Once the call is completed, the desired member of the structure can be accessed as an offset from EAX, as can be seen in the code generated for the example:

```
 struct test_tag {
            int  a;
            int  some_array[1  ];
            }  test_struct;

 struct test_tag test_function(struct test_tag test_parm)
{
    test_parm.a = 42;
   return  test_parm;
}

int  main(void)
{
   return  test_function(test_struct).a;
}
```

The code generated for the example would be:

```
   PUBLIC    main
main    PROC
   PUSH       EBP
   MOV         EBP,  ESP
    SUB         ESP,  194H        ; Allocate space for compiler-generated
   PUSH       ESI                  ;    temporary  variable
```

```
    PUSH     EDI
    SUB      ESP,  194H
    MOV      EDI,  ESP
    MOV      ESI, OFFSET FLAT: test_struct
    MOV      ECX,  65H
    REP  MOVSD
    LEA      EAX,  [ESP+ 194H]
    PUSH     EAX
    CALL     TEST_FUNCTION
    MOV      EAX, [EAX]         ; Note the convenience of having the
    POP      EDI                ;   address of the returned structure
    POP      ESI                ;   in  EAX
    LEAVE
    RET
main    ENDP
```

Examples Using the _Pascal Convention

Chapter 15. Developing Virtual Device Drivers

The C/C++ Tools compiler provides a number of features specifically for virtual device driver development. This chapter describes those features and discusses the issues you should be aware of when developing virtual device drivers. Note that support for developing virtual device drivers is available for C programs only.

Virtual device drivers (VDDs) provide virtual hardware support for DOS and DOS applications. They emulate input/output port and device memory operations. To achieve a certain level of hardware independence, a virtual device driver usually communicates with a physical device driver to interact with hardware. For example, the OS/2 operating system provides both virtual and physical device drivers for the mouse and keyboard.

User-supplied virtual device drivers simulate the hardware interfaces of an option adapter or device, and are usually used to migrate existing DOS applications into the OS/2 DOS environment.

A virtual device driver is essentially a DLL. It is responsible for presenting a virtual copy of the hardware resource to the DOS session and for coordinating physical access to that resource.

You may need to create a virtual device driver if multiple sessions must share access to a device where the input and output is not based on file handles, or if the particular device requires that interrupts be serviced within a short period of time.

For more information about virtual device drivers and how to create them, see the *Virtual Device Driver Reference* from the OS/2 2.0 Technical Library (10G3356). There is also a sample virtual device driver program included in the Toolkit.

Creating Code to Run at Ring Zero

Most object code runs at ring 3. However, some object code, such as that for virtual device drivers and operating systems, must run at ring . To generate code to run at ring , use the /Gr+ option. Note that to use /Gr+, you must also specify the /Rn option and use the subsystem libraries.

When you use the /Gr+ option, the compiler keeps track of which storage references are to the stack segment and which references are to the data segment, and ensures that the generated code is correct for these operations. This tracking is necessary because at ring , the stack segment and data segment may not be the same. (At ring 3, they are the same.)

In some cases, the compiler cannot tell whether the reference is to the stack or data segment. Usually the reason is that the control flow of the program allows for either possibility, depending on which path through the program is taken at run time. For this reason, when you take the address of a stack-based variable (such as a local variable or parameter), you cannot safely pass the address to another function. In addition, you cannot safely store a stack address and a static or external address in the same variable, and subsequently de-reference the pointer created by the operation.

Whenever you take the address of a stack-based variable, the compiler generates a warning message that the address might be used in an unsafe way. This message is not generated if you specify /Gr-.

If your VDD contains any functions that are called from 16-bit physical device drivers, you must compile them with the /Gv+ option to ensure the DS and ES registers are handled correctly. These two registers contain the selector for a 16-bit data segment. Using /Gv+ ensures that DS and ES are saved on entry to an external function, set to the selector for DGROUP, and then restored on exit from the function.

| **Note:** When you use /Gv+, if you also use the intermediate code linker (with the /Fw+ or /Ol+ option), only use the /Gu+ option if the functions affected by /Gv+ are explicitly exported. If they are not exported, do not use the /Gu+ option. Because of this restriction, using the intermediate code linker for this type of program may not greatly improve the optimization of your code.

Using Virtual Device Driver Calling Conventions

If you are building a VDD in C, you must use 32-bit _Pascal or _Far32 _Pascal calling conventions to call the Virtual Driver Help interfaces or communicate with physical device drivers. These calling conventions are not supported for C++ programs. Within a VDD, you can use the _Optlink convention in most cases. Private interfaces between physical and virtual device drivers can use any calling convention provided both device drivers support it.

The _Far32 _Pascal calling convention is only available for code running at ring . It allows you to make calls between code segments with different selectors. It also allows your VDDs to communicate with physical device drivers.

You can specify the calling convention using either the _Pascal and _Far32 _Pascal keywords or the #pragma linkage directive. The description of the implementation of the _Pascal calling conventions is in "_Pascal and _Far32 _Pascal Calling Conventions" on page 272.

Using _Far32 _Pascal Function Pointers

The C/C++ Tools compiler provides special 48-bit function pointers so you can make indirect calls to 32-bit functions that use th _Far32 _Pascal convention. The _Far32 _Pascal pointers are required to build VDDs and similar applications that run at ring . For example, you would use 48-bit pointers to allow your VDD to communicate with physical device drivers.

The _Far32 _Pascal pointers, like the _Far32 _Pascal calling convention, are only supported when the /Gr+ option is specified.

_Far32 _Pascal Function Pointers

The 48-bit pointer consists of 2 fields:

1. A 16-bit selector value which identifies the code segment
2. A 32-bit offset value which identifies the function's location in the segment.

To declare a 48-bit pointer, use the _Far32 and _Pascal keywords in the pointer declaration. For example:

```
void ( _Far32 _Pascal foo)(int);
```

declares foo to be a 48-bit pointer to a function with the _Far32 _Pascal convention that takes an integer argument and does not return a value.

The only operations that can be performed on or with a _Far32 _Pascal pointer are:

- Calling the function.
- Assigning the pointer, which includes casting it to a 32-bit function pointer or to an integer or unsigned type.
- Initializing the pointer, either statically or at runtime, with the address of a _Far32 _Pascal or 32-bit function, or with an integer or unsigned value.
- Comparing two pointers for equality or inequality. Like all function pointers, 48-bit pointers cannot be compared using relational operators.
- Passing the pointer as a parameter or returning it from a function. 48-bit pointers are passed in the same way as aggregates. The offset portion is returned in EAX and the segment portion in DX.

If you assign an integer or unsigned value to a 48-bit pointer, the selector field of the pointer is set to the default CODE32 segment, and the offset field is initialized to the integer value being assigned. This type of assignment is not generally useful, because you cannot know where a function will reside in a code segment, and because if your code segment is CODE32, a 32-bit function pointer is sufficient.

Note: _Far32 _Pascal pointers cannot be directly converted to _Far16 pointers.

Creating a Module Definition File

When you link your VDD, you must use a module definition (.DEF) file. The first statement in the file must be

VIRTUAL DEVICE *device_name*

where *device_name* specifies the name of the VDD. The file cannot contain a NAME statement.

Once you have created your device driver, you must place a DEVICE statement in your CONFIG.SYS file to ensure it is treated as a device by the operating system.

For more details on .DEF file statements, see the Toolkit online *Tools Reference*. For additional information on writing and building device drivers, see the online *Control Program Reference* and the Technical Library *Virtual Device Driver Reference*.

VDD Module Definition Files

Chapter 16. Calling Between 32-Bit and 16-Bit Code

This chapter discusses how to call 16-bit code from your 32-bit C/C++ Tools programs and how to call back to your program from the 16-bit code. If you have applications that depend on APIs or subroutines that are only available as 16-bit code, or have developed or purchased libraries of routines that are currently 16-bit code, you will need to call this 16-bit code from C/C++ Tools code.

Note: The C/C++ Tools compiler produces 32-bit code **only**. It does not produce 16-bit code.

This chapter describes the 16-bit calling conventions supported by the C/C++ Tools product, as well as how to share and pass pointers and data between 32-bit and 16-bit code. The conventions and methods described apply for both C and C++ programs.

Note: Before calling 16-bit object modules, you must know the calling convention that object code uses and use the same convention.

You can statically link between 32-bit and 16-bit code with the following restrictions:

- The main function must be 32-bit code.
- You cannot make any calls to 16-bit library functions in the 16-bit code.
- You must compile the 16-bit code with the /ND option (with a 16-bit compiler).

These restrictions do not apply when you dynamically link 32-bit code to 16-bit DLLs.

Declaring 16-Bit Functions

There are three calling conventions for calling 16-bit code:

```
_Far16 _Cdecl
_Far16 _Fastcall
_Far16 _Pascal
```

The _Far16 _Cdecl and _Far16 _Pascal conventions are equivalent to the cdecl and pascal conventions used in the IBM C/2* and Microsoft** C Version 6.0 compilers. The _Far16 _Fastcall convention is equivalent to the Microsoft C Version 6.0 fastcall convention.

For details on how these calling conventions work and how they differ from each other, see “Understanding 16-Bit Calling Conventions” on page 297.

You can specify the calling convention for a function using linkage keywords or, in C programs only, the #pragma linkage directive. For example, the following fragment uses keywords to declare the function dave as a 16-bit function using the _Far16 _Pascal calling convention:

```
int _Far16 _Pascal dave(short, char );
```

Note: It is good programming practice to include a prototype for 16-bit functions, but it is not necessary.

You can also specify the stack size for 16-bit code using the #pragma stack16 directive. For example, the following directive sets the stack size to 8192 bytes (8K):

```
#pragma stack16(8192)
```

The default stack size is 4096 bytes (4K). This size is used for all 16-bit functions called after the #pragma directive until the end of the compilation unit, or until another #pragma stack16 is encountered. Note that the 16-bit stack is allocated from the 32-bit stack, so you must ensure that the 32-bit stack is large enough for both your 32-bit and 16-bit code.

For more information on #pragma linkage, #pragma stack16, and the linkage keywords, see the *Online Language Reference*.

Declaring Segmented Pointers

Because pointers have a different format in 16-bit code than they do in 32-bit code, sharing or passing them between 32-bit and 16-bit code requires certain actions. Use the _Seg16 type qualifier to declare external pointers that will be shared between 32-bit and 16-bit code, that is, that are declared in both. For example:

```
char   _Seg16 p16;
```

directs the compiler to store the pointer as a segmented pointer (with a 16-bit selector and 16-bit offset) that can be used directly by a 16-bit application. You can also use this pointer in a 32-bit program; the C/C++ Tools compiler automatically converts it to a flat 32-bit pointer when necessary.

Note: The _Seg16 keyword comes **after** the asterisk in the declaration, as required by ANSI syntax rules. Programmers familiar with the IBM C/2 and Microsoft C Version 6.0 compilers may be used to placing the far keyword in their declarations, but to the left of the asterisk:

```
char far   x;
```

Because this syntax is contrary to ANSI binding rules, the C/C++ Tools product does not support it.

Not all pointers passed to 16-bit functions need to be qualified with _Seg16. If the pointer is passed to the function as a member of an aggregate or an array, you must qualify it with _Seg16. The _Seg16 keyword is also required if you are using two or more levels of indirection (for example, a pointer to a pointer). If the pointer is passed directly as a parameter, the compiler automatically converts it to a 16-bit pointer and the _Seg16 keyword is not required. However, if your pointers are used primarily as parameters to 16-bit functions and are not used extensively in your 32-bit code, it may be advantageous to declare them with _Seg16.

Use the _Seg16 qualifier only when necessary. Because of the conversions that are performed whenever a _Seg16 pointer is used in 32-bit code, unnecessary use of segmented pointers can cause a noticeable degradation in the performance of your application.

Declaring Shared Objects

Because a 16-bit program cannot access a data item that is larger than 64K in size or that spans a 64K boundary in memory, any data items that are to be shared between 16-bit and 32-bit programs must conform to these limits. Use the #pragma seg16 directive to ensure that shared data items do not cross 64K boundaries. In most cases, you need only use this #pragma directive with items that are likely to cross 64K boundaries, such as aggregates, doubles, and long doubles.

You can use #pragma seg16 either with the data item directly or through a typedef. The following code fragment shows both ways of using #pragma seg16:

```
struct family {
  long              john;
  double            carolynn;
   char  _Seg16 geoff;
  long              colleen;
};

#pragma seg16( cat )
struct family cat;               / cat is qualified directly /

typedef struct family tom;             1
#pragma seg16( tom )                        2

tom    edna;             / edna is qualified using a typedef /  3
```

Note: Using #pragma seg16 on variables of type struct family does not mean that pointers inside the structure will automatically be qualified with _Seg16. If you want the pointers to be qualified as such, you must declare them yourself.

The #pragma seg16 directive can be used either before or after the variable or typedef name is declared. In the case of the typedef, however, the #pragma must be attached to the typedef name before that name is used in another declaration. For example, in the preceding example, the lines marked 1 and 2 can appear in any order, but both must appear before the line marked 3 .

Because data objects used in 16-bit programs must be smaller than 64K in size, the #pragma seg16 directive cannot be used on objects greater than 64K.

Converting Structures

If a structure will be referenced in both 32-bit and 16-bit code and contains bit-fields or members of type int or enum, you may have to rewrite the structure to ensure that all members align properly.

16-bit compilers define type int with a different size than the C/C++ Tools compiler. To ensure all integers map the same way, change your integer declarations to use short for 2-byte integers and long for 4-byte integers.

The size of type enum also differs between compilers. For example, the C/2 compiler makes all enum types 2 bytes, while the C/C++ Tools compiler defines the size as 1, 2, or 4 bytes, depending on the range of values the enumeration contains. You can use the /Su option to force the C/C++ Tools compiler to make the size of an enum type 1, 2, or 4 bytes, or to use the SAA rules that make all enum variables the size of the smallest integral type that can contain all variables.

Bit fields are also mapped differently by different compilers. The C/C++ Tools compiler stores bit fields in the smallest number of bytes large enough to hold them. For a description of the C/C++ Tools bit-field mapping and alignment, see 399.

You may also need to pack your structures. See Appendix C, “Mapping” on page 385 for details of how the C/C++ Tools compiler aligns structure members. If the mapping performed by your 16-bit compiler differs, declare your structures as packed in both your 32-bit and 16-bit code.

Compiler Option for 16-Bit Declarations

The C/C++ Tools compiler also provides the /Gt compiler option to enable data to be shared between 32-bit and 16-bit code. When you compile a program with /Gt+, an implicit #pragma seg16 directive is performed for all variable declarations. Pointers are **not** implicitly qualified with _Seg16; you must qualify them if desired.

The /Gt+ option also defines special versions of the malloc family of functions that return memory that can be safely used by 16-bit code. When /Gt+ is specified, all calls to calloc, malloc, realloc, and free are mapped to _tcalloc, _tmalloc, _trealloc, and _tfree respectively.

These functions work exactly like the original functions, but the memory allocated or freed is guaranteed not to cross 64K boundaries, allowing the objects declared to be used in 16-bit programs. This memory is also called *tiled* memory. Tiled memory is limited to 512M per process.

Note: When you use the /Gt+ option, data items larger than 64K in size will be aligned on 64K boundaries, but will also cross 64K boundaries.

Calling Back to 32-Bit Code from 16-Bit Code

Some 16-bit applications require that calling applications register callback functions. For example, IBM Communications Manager requires callback functions to handle certain events. When you call these 16-bit applications from 32-bit code, you can pass a pointer to a 32-bit function that will act as the callback function.

The 32-bit callback function must use the _Far16 _Cdecl or _Far16 _Pascal calling convention. The _Far16 _Fastcall convention is not supported for callback functions. All pointer parameters must be qualified with the _Seg16 type qualifier.

The C/C++ Tools compiler performs all necessary changes from the 16-bit to the 32-bit environment on entry to the callback function, and from 32-bit to 16-bit on exit. Note that callback functions can only be called indirectly.

Restrictions on 16-Bit Calls and Callbacks

A function calling a 16-bit routine performs maintenance on its own stack to ensure that the stack will not cross a 64K boundary within the 16-bit routine. When the function has a variable-length argument list or no prototype statements, this stack maintenance does not occur, and the stack may cross 64K boundaries within the 16-bit routine. It is therefore unsafe to pass the address of a parameter or automatic variable to 16-bit code from one of these functions.

The compiler ensures that no parameters or automatic variables of a function calling 16-bit code cross a 64K boundary. Any parameters or automatic variables of functions that do not call 16-bit code may cross 64K boundaries. Passing the address of the parameters or automatic variables to functions that pass them on to 16-bit code will result in an unreliable program.

To work around this problem, copy the value passed into an automatic variable in the function that calls the 16-bit code. This automatic variable will not cross a 64K boundary.

Memory returned by _alloca will not be tiled. If a function contains a call to _alloca, it should not also call 16-bit code, because parameters and automatic variables may then cross 64K boundaries.

A 16-bit program cannot pass structures by value to a 32-bit callback function. The callback function cannot return structures by value to the 16-bit program that called it.

The parameter area of the callback function cannot be larger than 120 bytes.

Example of Calling a 16-Bit Program

The sample program SAMPLE04 shows how to call 16-bit code from a 32-bit program, and also how to call back to a 32-bit function from a 16-bit routine. The 16-bit code is placed in two DLLs, one of which is bound to the 32-bit program at load time by using IMPLIB to build an import library. The other is bound at run time using OS/2 APIs. When the program is run, it prints a stanza from a poem.

Although the source for the 16-bit routines is included in SAMPLE04 for demonstration purposes, the mechanisms used to call the routines can also be applied when the 16-bit source is not available.

Important: To compile, link, and run this example, you must have the IBM C/2 or Microsoft C Version 6.0 16-bit compiler installed, and its main directory must be included in the PATH statement of your CONFIG.SYS file.

The files for the sample program are:

SAMPLE 4.C The source file for the 32-bit program

SAMPLE 4.H The user include file

SAMPLE 4.DEF The module definition file for the 32-bit program

SAMP 4A.C The source file for the first 16-bit DLL (bound at load time)

SAMP 4A.DEF The module definition file for the first 16-bit DLL

SAMP 4B.C The source file for the second 16-bit DLL (bound at run time)

SAMP 4B.DEF The module definition file for the second 16-bit DLL.

Example of Calling a 16-Bit Program

The 32-bit main program (SAMPLE 4.C):

- Makes direct calls to the 16-bit functions plugh1 and plugh2, which are both defined in the 16-bit DLL bound at load time (the source for which is SAMP 4A.C).
- Demonstrates a callback function. The 32-bit user function xyzzy is passed to the 16-bit plugh3 routine (defined in SAMP 4A.C) with the intent that the 16-bit routine will then call the user function. The xyzzy function is declared using a 16-bit calling convention and is called from the 16-bit DLL, but it is run as a 32-bit function.
- Uses OS/2 APIs to load the runtime DLL (the source for which is SAMP 4B.C) and query the address of the function plugh4. The program then calls plugh4 using the function pointer returned by the API.

If you installed the sample programs, these files are found in the SAMPLES\SAMPLE04 directory under the main C/C++ Tools directory. Two make files that build the sample are also provided, MAKE 4S for static linking and MAKE 4D for dynamic linking.

Note: You must have the Toolkit installed to use the make files.

To compile and link this sample program, at the prompt in the SAMPLES\SAMPLE04 directory, use NMAKE with the appropriate make file. For example:

```
nmake all /f MAKE 4D
```

Example of Calling a 16-Bit Program

To compile and link the program yourself, use the following commands:

Command	Description
cl -c -Alfu -G2s SAMP 4A.C	Compiles the first 16-bit program. The options used are: -c Compile only. -Alfu Use large memory model. -G2s Use 80286 instructions; turn stack probes off.
link /MAP /NOI /NOD SAMP 4A,SAMP 4A.DLL,SAMP 4A,llibcdll os2286,SAMP 4A	Links the first 16-bit program to create a DLL. The link options are: /MAP Create a map file. /NOI Do not ignore case. /NOD Do not use default library names.
cl -c -Alfu -G2s SAMP 4B.C	Compiles the second 16-bit program.
link /MAP /NOI /NOD SAMP 4B,SAMP 4B.DLL,SAMP 4B,LLIBCDLL OS2286,SAMP 4B	Links the second 16-bit program to create a DLL.
icc /C SAMPLE 4.C	Compiles the 32-bit program. The /C option specifies compile only.
LINK386 /MAP /NOI /PM:vio SAMPLE 4,,SAMPLE 4,SAMP 4A,SAMPLE 4	Links the 32-bit program to create an executable module that is also linked to the SAMP 4A.DLL.

To run the program, enter SAMPLE 4.

Understanding 16-Bit Calling Conventions

There are three 16-bit calling conventions supported by the C/C++ Tools compiler: _Far16 _Cdecl, _Far16 _Fastcall, and _Far16 _Pascal. This section explains how these conventions work and how they differ from each other.

Similarities between the 16-Bit Conventions

The general rules for all three 16-bit calling conventions are:

- Types char, unsigned char, short, and unsigned short occupy a word on the stack.
- Types long and unsigned long occupy a doubleword with the value's high-order word pushed first.
- Types float, double, and long double are passed directly on the 80386 stack as 32-, 64-, and 80-bit values respectively.
- char types are sign-extended when expanded to word or doubleword size; unsigned char types are zero-extended on the stack.
- Far pointers are 32 bits and are pushed such that the segment value is pushed first and the offset second.
- If the argument is a structure, the last word is pushed first and each successive word is pushed until the first word.
- All arrays are passed by reference.
- BP, SI, and DI registers must be preserved across the call.
- Segment registers must be preserved across the call.
- Structures passed on the stack are rounded up in size to the next word boundary.
- The direction flag must be clear on entry and exit.

Return values are passed back to the caller as follows:

- Types char, unsigned char, short, and unsigned short are returned in AX.
- Types long and unsigned long are returned such that the high word is in DX and the low word is in AX.
- Far pointers are returned such that the offset is in AX and the selector is in DX.

Differences between the 16-Bit Conventions

When you use the _Far16 _Cdecl calling convention, the parameters are pushed on the stack in a right-to-left order. The caller cleans up the parameters on the stack. This is the opposite of the _Far16 _Pascal and _Far16 _Fastcall conventions. When you use the _Far16 _Pascal convention, the parameters are pushed on the stack from left to right, and the callee (the function being called) cleans up the stack (usually by using a RET nn where nn is the number of bytes in the parameter list).

The _Far16 _Fastcall convention differs from _Far16 _Cdecl and _Far16 _Pascal in that it uses three registers that can take parameters, similar to _Optlink. When you use _Far16 _Fastcall, registers are assigned to variable types as follows:

Types char and unsigned char are stored in AL, DL, and BL.
Types short and unsigned short are stored in AX, DX, and BX.
Types long and unsigned long are stored such that the high word is in DX and the low word is in AX.
All other types are passed on the stack.

Arguments are stored in the first available register allocated for their type. If all registers for that type are filled, the argument is pushed on the 80386 stack from left to right.

Another difference is the method of returning structures, unions, and floating-point types. For _Far16 _Cdecl and _Far16 _Pascal, all three types are returned with the address returned like a far pointer; that is, the value is in storage. The _Far16 _Pascal convention passes a hidden parameter, while _Far16 _Cdecl has a static area. This means that the _Far16 _Cdecl convention is nonreentrant, and should not be used in multithread programs. See “Return Values from 16-Bit Calls” for more details on how values are returned from 16-bit calls.

When you use the _Far16 _Fastcall convention, structures and unions are returned with the address returned like a near pointer. Like _Far16 _Pascal, _Far16 _Fastcall passes the address as a hidden parameter. Floating-point types are returned in ST(0).

Return Values from 16-Bit Calls

The following examples demonstrate how the C/C++ Tools compiler expects values to be returned from calls to 16-bit programs.

Note: This is the same way that the IBM C/2 and Microsoft C Version 6.0 compilers return values.

```
char cdecl myfunc(double,float,struct x);
char pascal myfunc(double,float,struct x);
char fastcall myfunc(double,float,struct x);

unsigned char cdecl myfunc(double,float,struct x);
unsigned char pascal myfunc(double,float,struct x);
unsigned char fastcall myfunc(double,float,struct x);
```

The returned value is placed in AL.

```
short cdecl myfunc(double,float,struct x);
short pascal myfunc(double,float,struct x);
short fastcall myfunc(double,float,struct x);
```

The returned value is placed in AX.

```
long cdecl myfunc(double,float,struct x);
long pascal myfunc(double,float,struct x);
long fastcall myfunc(double,float,struct x);
```

The high word is in DX, and the low word is in AX.

Return Values from 16-Bit Calls

```
float cdecl myfunc(double, float, struct x);
double cdecl myfunc(double, float, struct x);
long double cdecl myfunc(double, float, struct x);
```

The compiler does not provide a hidden parameter, but rather places the return value in an external static variable __fac, which is defined as a QWORD. On return, DX contains the selector and AX contains the offset of __fac.

For functions with type long double cdecl, the returned value is placed in ST(0).

```
float pascal myfunc(double,float,struct x);
double pascal myfunc(double,float,struct x);
long double pascal myfunc(double,float,struct x);
```

The compiler reserves space in automatic storage for the return value and pushes (last) a pointer to this area (offset only, SS is always assumed). The callee stores the return value in this area and returns the offset of this area in AX and returns SS in DX.

```
float fastcall myfunc(double,float,struct x);
double fastcall myfunc(double,float,struct x);
long double fastcall myfunc(double,float,struct x);
```

The returned value is placed in ST(0).

```
char far   cdecl myfunc(double,float,struct x);
char far   pascal myfunc(double,float,struct x)
char far   fastcall myfunc(double,float,struct x)
```

Far pointers are returned such that the offset is in AX and the selector is in DX.

```
struct_2 _bytes cdecl myfunc(double,float,struct x)
```

The compiler reserves sizeof(struct_2 _bytes) in uninitialized static (BSS) for the callee. No hidden parameter is passed; the callee moves the return structure into this static reserved area and returns the offset of the structure in AX and the selector in DX.

struct_2 _bytes pascal myfunc(double,float,struct x)
struct_2 _bytes fastcall myfunc(double,float,struct x)

The compiler reserves space for the return value in the caller's automatic storage and pushes the address of this area as a near pointer (SS will be assumed as the selector). This parameter is pushed last as a hidden parameter. The offset of the reserved space is returned in AX, and the selector (SS) is returned in DX.

struct_4_bytes cdecl myfunc(double,float,struct x)
struct_4_bytes fastcall myfunc(double,float,struct x)

The compiler returns the contents of the structure in AX and DX. AX contains the lower 2 bytes, and DX the higher 2 bytes.

- If the structure is packed and its size is 1 byte, AL is used.
- If the structure's size is 2 bytes, AX is used.
- If the structure is packed and its size is 3 bytes, space is reserved in the data segment, the offset is returned in AX, and the selector is returned in DX.

struct_4_bytes pascal myfunc(double,float,struct x)

The compiler reserves space for the return value in the caller's automatic storage and pushes the address of this area as a near pointer (SS will be assumed as the selector). This parameter is pushed last as a hidden parameter. The offset of the reserved space is returned in AX, and the selector (SS) is returned in DX.

char cdecl myfunc(double,float,struct x)
char pascal myfunc(double,float,struct x)
char fastcall myfunc(double,float,struct x)

unsigned char cdecl myfunc(double,float,struct x)
unsigned char pascal myfunc(double,float,struct x)
unsigned char fastcall myfunc(double,float,struct x)

The returned value is placed in AL.

Return Values from 16-Bit Calls

Chapter 17. Developing Subsystems

A subsystem is a collection of code and/or data that can be shared across processes and that does not use the C/C++ Tools runtime environment. This chapter describes how to create a subsystem.

A subsystem may have code and data segments that are shared by all processes, or it may have separate segments for each process. If the subsystem is a DLL, there is also an initialization routine associated with it.

By default, the C/C++ Tools compiler creates a runtime environment for you using C or C++ initializations, exception management, and termination. This environment allows runtime functions to perform input/output and other services. However, many applications require no runtime environment and must be written as subsystems. For example, you will want to turn off the runtime environment support to:

- Develop Presentation Manager display or printer drivers
- Develop virtual device drivers
- Develop installable file system drivers
- Create DLLs with global initialization/termination and a single automatic data segment that is shared by all processes. The initialization/termination function is called only once when the DLL is first loaded and when it is last freed.

Creating a Subsystem

To create a subsystem, you must first create one or more source files as you would for any other program. Subsystems can be written in C or C++. No special file extension is required.

When you do not use the runtime environment, you must provide your own initialization functions, multithread support, exception handling, and termination functions. You can use OS/2 APIs. For more information on the OS/2 APIs, see the Toolkit documentation.

If you need to pass parameters to a subsystem executable module, the argv and argc command-line parameters to main are supported. However, you cannot use the envp parameter to main.

Subsystem Library Functions

The libraries DDE4NBS.LIB and DDE4NBS.DLL are provided specifically for subsystem development. Use DDE4NBS.LIB for static linking, and DDE4NBS.DLL for dynamic linking. The import library DDE4NBSI.LIB is also provided for dynamic linking. You can also use the DDE4NBSO.LIB library to create your own subsystem runtime DLL. See "Creating Your Own Subsystem Runtime Library DLLs" on page 313 for more information on creating subsystem runtime DLLs.

Those C/C++ Tools library functions that require a runtime environment cannot be used in a subsystem. The subsystem libraries contain the library functions that do not require a runtime environment, including the extensions that allow low-level I/O. No other I/O functions are provided.

Note: The C++ I/O Stream Library is also available for subsystem development, as are the C++ runtime functions (new and delete) and exception handling functions (throw, try and catch). The Collection and User Interface class libraries are not available for subsystem development.

Subsystem Library Functions

The functions available in the subsystem libraries are:

abs	_filelength	qsort	strncmp
_access	_fpreset	_read	strncpy
_alloca	free	realloc	strpbrk
atof	_heapmin	remove	strrchr
atoi[1]	_isatty	rename	strspn
atol[1]	_itoa	setjmp[3]	strstr
bsearch	labs	_setmode	strtol
calloc	ldiv	_sopen	strtoul
_chmod	longjmp[3]	sprintf[4]	_tell
_chsize	_lseek	sscanf[4]	_ultoa
_clear87	_ltoa	_status87	_umask
_close	malloc	strcat	_unlink
_control87	memchr	strchr	va_arg[5]
_creat	memcmp	strcmp	va_end[5]
div	memcpy	strcpy	va_start[5]
_dup	memmove	strcspn	vprintf[4]
_dup2	memset	strdup	vsprintf[4]
__eof	_open	strlen	_write
exit[2]	printf[4]	strncat	

Notes:

1. The subsystem library versions of these functions do not use the locale information that the standard library versions use.
2. Note that atexit and _onexit are not provided.
3. You must write your own exception handler when using these functions in a subsystem.
4. When you use these functions in a subsystem, \n will be translated to \r\n and DosWrite will be used to write the contents of the buffer to stdout. There is no serialization protection and no multibyte support. These functions use only the default "C" locale information.
5. These functions are implemented as macros.

Calling Conventions for Subsystem Functions

When creating a subsystem, you can use either the _System or _Optlink convention for your functions. Any external functions that will be called from programs not compiled by the C/C++ Tools compiler **must** use the _System convention.

You can use the /Mp or /Ms options to specify the calling convention for all functions in a program, or you can use linkage keywords or the #pragma linkage directive to specify the convention for individual functions.

Note: The #pragma linkage directive is supported for C programs only.

Building a Subsystem DLL

To create a subsystem DLL, you can follow the same steps for building a DLL that uses the runtime environment, as described in Chapter 12, "Building Dynamic Link Libraries" on page 195. The _DLL_InitTerm function provided in the subsystem libraries differs from the runtime version.

The initialization and termination entry point for all DLLs is the _DLL_InitTerm function. In the C runtime environment, _DLL_InitTerm initializes and terminates the necessary environment for the DLL, including storage, semaphores, and variables. The version provided in the subsystem libraries defines the entry point for the DLL, but provides no initialization or termination functions.

If your subsystem DLL requires any initialization or termination, you will need to create your own _DLL_InitTerm function. Otherwise, you can use the default version.

Writing Your Own Subsystem _DLL_InitTerm Function

The prototype for the _DLL_InitTerm function is:

```
unsigned long _System _DLL_InitTerm(unsigned long modhandle,
                                    unsigned long flag);
```

If the value of the *flag* parameter is 0, the DLL environment is initialized. If the value of the *flag* parameter is 1, the DLL environment is ended.

The *modhandle* parameter is the module handle assigned by the operating system for this DLL. The module handle can be used as a parameter to various OS/2 API calls. For example, DosQueryModuleName can be used to return the fully qualified path name of the DLL, which tells you where the DLL was loaded from.

The return code from _DLL_InitTerm tells the loader if the initialization or termination was performed successfully. If the call was successful, _DLL_InitTerm returns a nonzero value. A return code of indicates that the function failed. If a failure is indicated, the loader will not load the program that is accessing the DLL.

Because it is called by the operating system loader, the _DLL_InitTerm function must be declared as having the _System calling convention.

You do not need to call _CRT_init and _CRT_term in your _DLL_InitTerm function, because there is no runtime environment to initialize or terminate. However, if you are coding in C++, you do need to call __ctordtorInit at the beginning of _DLL_InitTerm to correctly initialize static constructors and destructors, and __ctordtorTerm at the end to correctly terminate them.

If you change your DLL at a later time to use the regular runtime libraries, you must add calls to _CRT_init and _CRT_term, as described in "Writing Your Own _DLL_InitTerm Function" on page 209, to ensure that the runtime environment is correctly initialized.

Example of a Subsystem _DLL_InitTerm Function

The following figure shows the _DLL_InitTerm function for the sample program SAMPLE05. In the sample program, this function is included in the SAMPLE 5.C source file. You could also make your _DLL_InitTerm function a separate file. Note that this figure shows only a fragment of SAMPLE 5.C and not the entire source file.

```
/ _DLL_InitTerm() - called by the loader for DLL
initialization/termination   /
/ This function must return a non-zero value if successful and a zero value   /
/ if unsuccessful.                                                             /

unsigned long _DLL_InitTerm( unsigned long hModule, unsigned long ulFlag )
   {
   APIRET  rc;

    / If ulFlag is zero then initialization is required:                    /
    /      If the shared memory pointer is NULL then the DLL is being loaded    /
    /      for the first time so acquire the named shared storage for the      /
    /      process control structures.   A linked list of process control      /
    /      structures will be maintained.   Each time a new process loads this   /
    /      DLL, a new process control structure is created and it is inserted   /
    /      at the end of the list by calling DLLREGISTER.                       /
   /                                                                            /
    / If ulFlag is 1 then termination is required:                            /
    /      Call DLLDEREGISTER which will remove the process control structure    /
    /      and free the shared memory block from its virtual address space.     /

    switch( ulFlag )
       {
       case   :
            if ( !ulProcessCount )
                {
```

Figure 22 (Part 1 of 2). _DLL_InitTerm Function for SAMPLE05

```
            /  Create the shared mutex semaphore.                                  /

            if ( ( rc = DosCreateMutexSem( SHARED_SEMAPHORE_NAME,
                                            &hmtxSharedSem,
                                            ,
                                            FALSE ) ) != NO_ERROR )
               {
               printf( "DosCreateMutexSem rc = %lu\n", rc );
               return   ;
               }
            }

         /  Register the current process.                                        /

         if ( DLLREGISTER( ) )
            return   ;

         break;

      case  1:
         /  De-register the current process.                                     /

         if ( DLLDEREGISTER( ) )
            return   ;

         break;

      default:
         return   ;
      }

   /  Indicate success.    Non-zero means success!!!
/

   return  1;
   }
```

Figure 22 (Part 2 of 2). _DLL_InitTerm Function for SAMPLE05

Compiling Your Subsystem

To compile your source files into a subsystem, use the /Rn compiler option. When you use this option, the compiler does not generate the external references that would build an environment. The subsystem libraries are also specified in each object file to be linked in at link time. The default compiler option is /Re, which creates an object with a runtime environment.

If you are creating a subsystem DLL, you must use the /Ge- option in addition to /Rn.

You can use either static linking (/Gd-), which is the default, or dynamic linking (/Gd+).

Restrictions When You Are Using Subsystems

If you are creating an executable module, the envp parameter to main is not supported. However, the argv and argc parameters are available. See "Passing Data to a Program" on page 139 for a description of envp under the runtime environment.

The low-level I/O functions allow you to perform some input and output operations. You are responsible for the buffering and formatting of I/O.

Example of a Subsystem DLL

The sample program SAMPLE05 shows how to create a simple subsystem DLL and a program to access it.

The DLL keeps a global count of the number of processes that access it, running totals for each process that accesses the subsystem, and a grand total for all processes. There are two external entry points for programs accessing the subsystem. The first is DLLINCREMENT, which increments both the grand total and the total for the calling process by the amount passed in. The second entry point is DLLSTATS, which prints out statistics kept by the subsystem, including the grand total and the total for the current process.

The grand total and the total for the process are stored in a single shared data segment of the subsystem. Each process total is stored in its own data segment.

The files for the sample program are:

SAMPLE 5.C The source file to create the DLL.

SAMPLE 5.DEF The module definition file for the DLL.

SAMPLE 5.H The user include file.

MAIN 5.C The main program that accesses the subsystem.

MAIN 5.DEF The module definition file for MAIN 5.C.

If you installed the sample programs, these files are found in the SAMPLE\SAMPLE05 directory under the main C/C++ Tools directory. Two make files that build the sample are also provided, MAKE 5S for static linking and MAKE 5D for dynamic linking.

Note: You must have the Toolkit installed to use the make files.

To compile and link this sample program, at the prompt in the SAMPLES\SAMPLE05 directory, use NMAKE with the appropriate make file. For example:

```
nmake all /f MAKE 5S
```

Example of a Subsystem DLL

To compile and link the program yourself, use the following commands:

Command	Description
icc /O+ /Rn /Ge- SAMPLE 5.C SAMPLE 5.DEF	Compiles and links SAMPLE 5.C using the default options and: Turning optimization on (/O+) Using subsystem libraries (/Rn) Creating a DLL (/Ge-).
icc /O+ MAIN 5.C MAIN 5.DEF	Compiles and links MAIN 5.C using the default options and turning optimization on. **Note:** Because MAIN 5.C calls getchar, it must be compiled using the regular runtime libraries.

To run the program:

1. Copy the subsystem DLL to a directory that is specified in the LIBPATH statement of your CONFIG.SYS file.
2. Start the main program in one or more different OS/2 sessions by entering the command

   ```
   MAIN 5
   ```

3. Enter 1 to increment the counts or 2 to print the statistics in any process that you have started. Repeat this step as often as you want.
4. Enter x in each OS/2 session to terminate each process.

Creating Your Own Subsystem Runtime Library DLLs

If you are shipping your application to other users, you can use one of two methods to make the C/C++ Tools subsystem library functions available to the users of your application:

1. Statically bind every module to the library (.LIB) files.

   This method increases the size of your modules and also slows the performance because the DLL environment has to be initialized for each module.

2. Create your own runtime DLLs.

   This method provides one common DLL environment for your entire application. It also lets you apply changes to the runtime library without relinking your application, meaning that if the C/C++ Tools DLLs change, you need only rebuild your DLL.

To create your own subsystem runtime library, follow these steps:

1. Copy and rename the C/C++ Tools DDE4NBS.DEF file, for example to mysdll.def. You must also change the DLL name on the LIBRARY line of the .DEF file. DDE4NBS.DEF is installed in the LIB subdirectory under the main C/C++ Tools installation directory.

2. Remove any functions you do not use directly or indirectly from your .DEF file, including the STUB line. Do not delete anything with the comment next to it; variables and functions indicated by this comments are always required because they are called by startup functions.

3. Create a source file for your DLL, for example, mysdll.c. If you are creating a runtime library that contains only C/C++ Tools functions, create an empty source file. If you are adding your own functions to the library, put the code for them in this file.

4. Compile and link your DLL files. Use the /Ge- option to create a DLL and the /Rn option to create a subsystem. For example:

```
icc /Ge- /Rn mysdll.c mysdll.def
```

5. Use the IMPLIB utility from the Toolkit to create an import library for your DLL, as described in “Using Your DLL” on page 205. For example:

```
IMPLIB /NOI mysdlli.lib mysdll.def
```

6. Use the WorkFrame/2 LIB utility to add the object modules that contain the initialization and termination functions to your import library. These objects are needed by all executable modules and DLLs, are contained in DDE4NBSO.LIB for subsystem programs. See the WorkFrame/2 online documentation for information on how to use LIB.

   Note: If you do not use the WorkFrame/2 LIB utility, you must ensure that all objects that access your runtime DLL are statically linked to the appropriate object library. The compile and link commands are described in the next step.

7. Compile your executable modules and other DLLs with the /Gn+ option to exclude the default library information. For example:

```
icc /C /Gn+ /Ge+ /Rn myprog.c
icc /C /Gn+ /Ge- /Rn mydll.c
```

   When you link your objects, specify your own import library. If you are using or plan to use OS/2 APIs, specify OS2386.LIB also. For example:

```
LINK386 myprog.obj,,, mysdlli.lib OS2386.LIB
LINK386 mydll.obj,,, mysdlli.lib OS2386.LIB
```

   To compile and link in one step, use the commands:

```
icc /Gn+ /Ge+ /Rn myprog.c mysdlli.lib OS2386.LIB
icc /Gn+ /Ge- /Rn mydll.c mysdlli.lib OS2386.LIB
```

Note: If you did not use the WorkFrame/2 LIB utility to add the initialization and termination objects to your import library, when you link your modules, specify:

a. DDE4NBSO.LIB
b. Your import library
c. OS2386.LIB (to allow you to use OS/2 APIs)
d. The linker option /NOD.

For example:

```
LINK386 /NOD myprog.obj,,,DDE4NBSO.LIB mysdlli.lib OS2386.LIB;
LINK386 /NOD mydll.obj,,,DDE4NBSO.LIB mysdlli.lib OS2386.LIB;
```

The /NOD option tells the linker to disregard the default libraries specified in the object files and use only the libraries given on the command line. If you are using icc to invoke the linker for you, the commands would be:

```
icc /B"/NOD" /Rn myprog.c DDE4NBSO.LIB mysdlli.lib OS2386.LIB
icc /Ge- /B"/NOD" /Rn mydll.c DDE4NBSO.LIB mysdlli.lib OS2386.LIB
```

The linker then links the objects from the object library directly into your executable module or DLL.

Creating Subsystem Runtime Library DLLs

Chapter 18. Signal and OS/2 Exception Handling

The C/C++ Tools product and the OS/2 operating system both have the capability to detect and report runtime errors and abnormal conditions.

Abnormal conditions can be reported to you and handled in one of the following ways:

1. Using C/C++ Tools signal handlers. Error handling by signals is defined by the SAA and ANSI C standards and can be used in both C and C++ programs.
2. Using OS/2 exception handlers. The C/C++ Tools library provides a C-language OS/2 exception handler, _Exception, to map OS/2 exceptions to C signals and signal handlers. You can also create and use your own exception handlers.
3. Using C++ exception handling constructs. These constructs belong to the C++ language definition and can only be used in C++ code. C++ exception handling is described in detail in the *Online Language Reference*.

This chapter describes how to use signal handlers and OS/2 exception handlers alone and in combination. Where appropriate, the interaction between C++ exception handling and the handling of signals and OS/2 exceptions is also described. Both signal and OS/2 exception handling are implemented in C++ as they are in C. OS/2 exceptions and exception handlers are also described in the Toolkit documentation.

Note: The terms *signal*, *OS/2 exception*, and *C++ exception* are not interchangeable. A signal exists only within the C and C++ languages. An OS/2 exception is generated by the operating system, and may be used by the C/C++ Tools library to generate a signal. A C++ exception exists only within the C++ language. In this chapter, the term *exception* refers to an OS/2 exception unless otherwise specified.

Handling Signals

Signals are C and C++ language constructs provided for error handling. A signal is a condition reported as a result of an error in program execution. It may also be caused by deliberate programmer action. With the C/C++ Tools product, operating system exceptions are mapped to signals for you. The C/C++ Tools product provides a number of different symbols to differentiate between error conditions. The signal constants are defined in the <**signal.h**> header file.

C provides two functions that deal with signal handling in the runtime environment: raise and signal. Signals can be reported by an explicit call to raise, but are generally reported as a result of a machine interrupt (for example, division by zero), of a user action (for example, pressing Ctrl-C or Ctrl-Break), or of an operating system exception.

Use the signal function to specify how to handle a particular signal. For each signal, you can specify one of 3 types of handlers:

1. SIG_DFL

   Use the C/C++ Tools default handling. For most signals, the default action is to terminate the process with an error message. SeeFigure 23 on page 320 for a list of signals and the default action for each. If the /Tx+ option is specified, the default action can be accompanied by a dump of the machine state to file handle 2, which is usually associated with stderr. Note that you can change the destination of the machine-state dump and other messages using the _set_crt_msg_handle function, which is described in the *C Library Reference*.

2. SIG_IGN

   Ignore the condition and continue running the program. Some signals cannot be ignored, such as division by zero. If you specify SIG_IGN for one of these signals, the C/C++ Tools library will treat the signal as if SIG_DFL was specified.

3. Your own signal handler function

   Call the function you specify. It can be any function, and can call any library function. Note that when the signal is reported and your function is called, signal handling is reset to SIG_DFL to prevent recursion should the same signal be reported from your function.

The initial setting for all signals is SIG_DFL, the default action.

The signal and raise functions are described in more detail in the *C Library Reference*.

Default Handling of Signals

The runtime environment will perform default handling of a given signal unless a specific signal handler is established or the signal is disabled (set to SIG_IGN). You can also set or reset default handling by coding:

signal(*sig*, SIG_DFL);

The default handling depends upon the signal that is being handled. For most signals, the default is to pass the signal to the next exception handler in the chain (the chaining of exception handlers is described in “Registering an OS/2 Exception Handler” on page 344). Unless you have set up your own exception handler, as described in “Creating Your Own OS/2 Exception Handler” on page 334, the default OS/2 exception handler receives the signal and performs the default action, which is to terminate the program and return an exit code. The exit code indicates:

1. The reason for the program termination. See DosExecPgm in the Toolkit online *Control Program Reference* for the possible values and meanings of the termination code.
2. The return code from DosExit. See the Toolkit online *Control Program Reference* for the DosExit return codes.

Default Signal Handling

The following table lists the C signals that the C/C++ Tools runtime library supports, the source of the signal, and the default handling performed by the library.

Figure 23 (Page 1 of 2). Default Handling of Signals

Signal	Source	Default Action
SIGABRT	Abnormal termination signal sent by the abort function	Terminate the program with exit code 3.
SIGBREAK	Ctrl-Break signal	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
SIGFPE	Floating-point exceptions that are not masked[5], such as overflow, division by zero, integer math exceptions, and operations that are not valid	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
SIGILL	Disallowed instruction	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
SIGINT	Ctrl-C signal	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
SIGSEGV	Attempt to access a memory address that is not valid	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
SIGTERM	Program termination signal sent by the user or operating system	Pass the signal to the next exception handler in the chain. If the exception handler is the OS/2 handler, the program terminates.
SIGUSR1	User-defined signal	Ignored.

Figure 23 (Page 2 of 2). Default Handling of Signals

Signal	Source	Default Action
SIGUSR2	User-defined signal	Ignored.
SIGUSR3	User-defined signal	Ignored.

Establishing a Signal Handler

You can establish or register your own signal handler with a call to the signal function:

signal(*sig*, *sig_handler*);

where *sig_handler* is the address of your signal handling function. The signal handler is a C function that takes a single integer argument (or two arguments for SIGFPE), and may have either _System or _Optlink linkage.

A signal handler for a particular signal remains established until one of the following occurs:

- A different handler is established for the same signal.
- The signal is explicitly reset to the system default with the function call signal(*sig*, SIG_DFL);.
- The signal is reported. When your signal handler is called, the handling for that signal is reset to the default as if the function call signal(sig_num, SIG_DFL) were explicitly made immediately before the signal handler call.

Note: A signal handler can also become deregistered if the load module where the signal handler resides is deleted using the _freemod function. If this situation arises, when the signal is raised, an OS/2 exception occurs and the behavior is undefined.

5 For more information on masking floating-point exceptions, see“Handling Floating-Point Exceptions” on page 354 .

Writing a Signal Handler Function

A signal handler function has no limitations and may call any C library function. Your signal handler may handle the signal in any of the following ways:

1. Calling exit or abort to terminate the process. The behavior of these two functions does not change when they are called from a signal handler.
2. Calling _endthread to terminate the current thread of a multithread program. The process continues to run without the thread. You must ensure that the loss of the thread does not affect the process. Note that calling _endthread for thread 1 of your process is the same as calling exit.
3. Calling longjmp to continue running the current thread from a known point. To call longjmp, you must have previously called setjmp in the current thread. The setjmp function saves the state of the thread in a buffer. When you call longjmp, the state of the thread is reset to the state in the buffer and the thread restarts at the call to setjmp.
4. Returning from the function to restart the thread as though the signal has not occurred. If this is not possible, the C/C++ Tools library terminates your process.

Example of a C Signal Handler

The following code gives a simple example of a signal handler function for a single-thread program. In the example, the function chkptr checks a given number of bytes in an area of storage and returns the number of bytes that you can access. The flow of the function's execution is described after the code.

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

static void mysig(int sig);              /  signal handler prototype  /
static jmp_buf jbuf;                     /  buffer for machine state  /

int chkptr(void   ptr, int size)
{
    void (  oldsig)(int);                /  where to save the old signal handler  /
    volatile char c;                     /  volatile to ensure access occurs  /
    int valid =  ;                       /  count of valid bytes  /
    char   p = ptr;

    oldsig = signal(SIGSEGV,mysig); /  set the signal handler  /              1

    if (!setjmp(jbuf))                   /  provide a point for the         /   2
    {                                    /     signal handler to return to  /

        while  (size--)
        {
            c =  p++;                    /  check the storage and  /              3
            valid++;                     /     increase the counter  /
        }
    }
```

Figure 24 (Part 1 of 2). Example Illustrating a Signal Handler

```
     signal(SIGSEGV,oldsig);              / reset the signal handler /         5
     return valid;                       / return number of valid bytes /    6
}

static void mysig(int sig)
{
     printf("Detected storage address that is not valid.\n");                   4
     longjmp(jbuf,1);            / return to the point of the setjmp call /

}
```

Figure 24 (Part 2 of 2). Example Illustrating a Signal Handler

1 The program registers the signal handler mysig and saves the original handler in oldsig so that it can be reset at a later time.

2 The call to setjmp saves the state of the thread in jbuf. When you call setjmp directly, it returns , so the code within the if statement is run.

3 The loop reads in and checks each byte of the buffer, incrementing the valid count for each byte successfully copied to c.

Assuming that not all of the buffer space is available, at some point in the loop p points to a storage location the process cannot access. An OS/2 exception is generated and translated by the C/C++ Tools library to the SIGSEGV signal. The library then resets the signal handler for SIGSEGV to SIG_DFL and calls the signal handler registered for SIGSEGV (mysig).

4 The mysig function prints an error message and uses longjmp to return to the place of the setjmp call in chkptr.

Note: mysig does not reset the signal handler for SIGSEGV, because that signal is not intended to occur again. In some cases, you may want to reset signal handling before the signal handler function ends.

5 Because setjmp returns a nonzero value when it is called through longjmp, the if condition is now false and execution falls through to this line. The signal handling for SIGSEGV is reset to whatever it was when chkptr was entered.

6 The function returns the number of valid bytes in the buffer.

As the preceding example shows, your program can recover from a signal and continue to run with no related problems.

Signal Handling in Multithread Programs

Each thread has its own independent set of signals. If you establish a signal handler on one thread, it handles only signals generated on that thread. Conversely, all signals generated on a particular thread are handled by the signal handler specified for that thread. If you establish a signal handler or raise a signal on one thread, you do not affect any other thread.

When a thread starts, all of its signal handlers are set to SIG_DFL. If you want any other signal handling for that thread, you must use signal to register it.

Three signals can only occur in thread 1: SIGINT, SIGBREAK, and SIGTERM. If you want to establish a signal handler for these signals, you must call signal in thread 1, which usually contains the main function.

When you call the raise function, the signal handler for that signal must be established on the thread where the call was made.

Note: You can use raise to signal your own conditions using the signals SIGUSR1, SIGUSR2, and SIGUSR3, which are provided for user signals. You can also use this function to generate signals to test your signal handlers.

Signal Handling Considerations

When you use signal handlers, keep a number of points in mind:

- You can register anything as a signal handler. It is up to you to make sure that you are registering a valid function.
- If your signal handler resides in a DLL, ensure that you change the signal handler when you unload the DLL. If you unload your DLL without changing the signal handler, no warnings or error messages are generated. When your signal handler gets called, your program will probably terminate. If another DLL has been loaded in the same address range, your program may continue but with undefined results.
- The SIGSEGV signal may occur for a condition other than a data pointer that is not valid. For example, it can also occur if an address pointer goes outside of your code segment. Your signal handler should not assume that SIGSEGV always implies an invalid data pointer.
- The SIGILL signal does not always occur when you use a pointer to call a function that is not valid. If the pointer points to a valid instruction stream, SIGILL is not raised.
- When you use longjmp to leave a signal handler, ensure that the buffer you are jumping to was created by the thread that you are in. Do not call setjmp from one thread and longjmp from another. The C/C++ Tools library checks the contents of the buffer for this condition and terminates the process if they are not valid.
- If you use console I/O functions, including gets and scanf, and a SIGINT, SIGBREAK, or SIGTERM signal occurs, the signal is reported **after** the library function returns. Because your signal handler can call any library function, one of these functions could be reentered. To protect the internal data structures, some library code is placed in "must complete" sections. When a signal occurs, the library waits until the "must complete" section ends before it reports the signal.

  Note: You can use the OS/2 APIs DosEnterMustComplete and DosExitMustComplete to create your own "must complete" sections of code. See the Toolkit documentation for more information on these APIs.

Variables referenced by both the signal handler and by other code should be given the attribute volatile to ensure they are always updated when they are referenced. Because of the way the compiler optimizes code, the following example may not work as intended when compiled with the /O+ option:

```
void sig_handler(int);
static int stepnum;

int main(void)
{
      stepnum =  ;
     signal(SIGSEGV,  sig_handler);

   ⋮
      stepnum = 1;      1

   ⋮
      stepnum = 2;      2
}

void sig_handler(int x)
{
      printf("Error at step %d\n", stepnum);
}
```

When using optimization, the compiler may not immediately store the value 1 for the variable stepnum. It may never store the value 1, and store only the value 2. If a signal occurs between statement 1 and statement 2, the value of stepnum passed to sig_handler may not be correct.

Declaring stepnum as volatile indicates to the compiler that references to this variable have side effects. Changes to the value of stepnum are then stored immediately.

C++ Consideration: When you use longjmp to recover from a signal in a C++ program, automatic destructors are not called for objects placed on the stack between the longjmp call and the corresponding setjmp call. Because the ANSI draft of the C++ language does not specify the behavior of a throw statement in a signal handler, the most portable way to ensure the appropriate destructors are called is to add statements to the setjmp location that will do a throw if necessary.

Handling OS/2 Exceptions

An OS/2 exception is generated by the operating system to report an abnormal condition. OS/2 exceptions are grouped into two categories:

1. Asynchronous exceptions, which are caused by actions outside of your current thread. There are only two:

   XCPT_SIGNAL, caused by a keyboard signal (Ctrl-C, Ctrl-Break) or the process termination exception. This exception can only occur on thread 1 of your process.

   XCPT_ASYNC_PROCESS_TERMINATE, caused by one of your threads terminating the entire process. This exception can occur on any thread.

2. Synchronous exceptions, which are caused by code in the thread that receives the exception. All other OS/2 exceptions fall into this category.

Just as you use signal handlers to handle signals, use exception handlers to handle OS/2 exceptions. Because signal handling is simpler than exception handling while exception handling offers additional function, you may want to use both in your program.

C/C++ Tools Default OS/2 Exception Handling

The C/C++ Tools library provides its own default exception handling functions: _Lib_excpt for OS/2 exceptions occurring in library functions and _Exception for all other OS/2 exceptions. You can use these exception handlers or create your own as described in "Creating Your Own OS/2 Exception Handler" on page 334.

The function _Exception is the C language exception handler. It is declared as:

```
#include <os2.h>

unsigned long _System _Exception(EXCEPTIONREPORTRECORD  report_rec,
                                 EXCEPTIONREGISTRATIONRECORD  reg_rec,
                                 CONTEXTRECORD  exc,
                                 void  dummy);
```

C/C++ Tools Default OS/2 Exception Handling

This exception handler is registered by the C/C++ Tools compiler for every thread or process. It maps recognized OS/2 exceptions to C signals, which can then be passed by the runtime library to the appropriate signal handlers.

Figure 25 shows which types of OS/2 exception are recognized by _Exception, the names of the exceptions, and the C signals to which each exception type is mapped. **These are the only OS/2 exceptions handled by _Exception**. The **Continuable** column indicates whether the program will continue if the corresponding signal handler is SIG_IGN or if a user-defined signal handler returns. If "No" is indicated, the program can only be continued if you provide a signal handler that uses longjmp to jump to another part of the program.

If the signal handler value is set to SIG_DFL, the default action taken for each of these exceptions is to terminate the program with an exit code of 99.

Figure 25 (Page 1 of 2). Mapping Between Exceptions and C Signals

OS/2 Exception	C Signal	Continuable?
Divide by zero XCPT_INTEGER_DIVIDE_BY_ZERO	SIGFPE	No
NPX387 error XCPT_FLOAT_DENORMAL_OPERAND XCPT_FLOAT_DIVIDE_BY_ZERO XCPT_FLOAT_INEXACT_RESULT XCPT_FLOAT_INVALID_OPERATION XCPT_FLOAT_OVERFLOW XCPT_FLOAT_STACK_CHECK XCPT_FLOAT_UNDERFLOW	SIGFPE	No; except for XCPT_FLOAT_INEXACT_RESULT
Overflow occurred XCPT_INTEGER_OVERFLOW	SIGFPE	Yes; resets the overflow flag
Bound opcode failed XCPT_ARRAY_BOUNDS_EXCEEDED	SIGFPE	No
Opcode not valid XCPT_ILLEGAL_INSTRUCTION XCPT_INVALID_LOCK_SEQUENCE XCPT_PRIVILEGED_INSTRUCTION	SIGILL	No

Figure 25 (Page 2 of 2). Mapping Between Exceptions and C Signals

OS/2 Exception	C Signal	Continuable?
General Protection fault XCPT_ACCESS_VIOLATION XCPT_DATATYPE_MISALIGNMENT	SIGSEGV	No
Ctrl-Break XCPT_SIGNAL (XCPT_SIGNAL_BREAK)	SIGBREAK	Yes
Ctrl-C XCPT_SIGNAL (XCPT_SIGNAL_INTR)	SIGINT	Yes
End process XCPT_SIGNAL (XCPT_SIGNAL_KILLPROC)	SIGTERM	Yes

Note: The Overflow and Bound opcode exceptions are provided for completeness only. They will never be caused by code generated by the C/C++ Tools compiler.

The following OS/2 exceptions are also recognized, but have no corresponding C signal. If one of these OS/2 exceptions occurs, it is passed to the next available exception handler, or if none is registered, it is passed by default to the operating system:

OS/2 Exception	Continuable?
Out of stack exception XCPT_GUARD_PAGE_VIOLATION	Yes
Synchronous process termination XCPT_PROCESS_TERMINATE	No
Asynchronous process termination XCPT_ASYNC_PROCESS_TERMINATE	No
Unwind target not valid XCPT_INVALID_UNWIND_TARGET	No

An out-of-stack exception occurs when the guard page of the stack is accessed. When the operating system encounters this exception, it automatically allocates a new guard page and the exception is continued. If there is not enough stack for the system to process the exception, the program is terminated.

For more information on guard page allocation and automatic stack growth, see "Controlling Stack Allocation and Stack Probes" on page 67.

OS/2 Exception Handling in Library Functions

There are two classes of library functions that require special exception handling: math functions and critical functions.

OS/2 exceptions occurring in all other library functions are treated as though they occurred in regular user code.

Math Functions

Before _Exception converts an OS/2 exception to a C signal, it first calls the C/C++ Tools library exception handler, _Lib_excpt. The _Lib_excpt function determines if the exception occurred in a math library function. The _Lib_excpt function is declared as follows:

```
#include <os2.h>

unsigned long _System _Lib_excpt(EXCEPTIONREPORTRECORD  report_rec,
                                 EXCEPTIONREGISTRATIONRECORD  reg_rec,
                                 CONTEXTRECORD  ecx,
                                 void  dummy);
```

If the exception does occur in a math function and it is a floating-point error, _Lib_excpt handles the exception and returns XCPT_CONTINUE_EXECUTION to the operating system to indicate the exception has been handled. Any signal handler function you may have established will **not** be called.

Important: If you are creating your own exception handler, it should first call _Lib_excpt to ensure that the exception did not occur in a library function.

If the cause of the OS/2 exception was not a floating-point error, the exception is returned to _Exception. The _Exception function then converts the OS/2 exception to the corresponding C signal and performs one of the following actions:

1. Terminates the process. If /Tx+ was specified, _Exception performs a machine-state dump to file handle 2, unless the exception was SIGBREAK, SIGINT, or SIGTERM, in which case the machine state is not meaningful.
2. Handles the exception and returns XCPT_CONTINUE_EXECUTION to the operating system.
3. Calls the signal handler function provided by you for that signal. A return from the signal handler results in either the return of XCPT_CONTINUE_EXECUTION to the operating system or the termination of the process as in the first action above.

Note: For more information about exception-handling return codes, refer to the Toolkit documentation.

Critical Functions

All nonreentrant functions are classified as critical functions. Most I/O and allocation functions, and those that begin or end threads or processes, fall in this class. The critical functions are:

atexit	_execv	fputc	malloc	_spawnlp
calloc	_execve	fputs	_onexit	_spawnlpe
_cgets	_execvp	fread	printf	_spawnv
clearerr	_execvpe	free	_putch	_spawnve
_cprintf	exit	freopen	_putenv	_spawnvp
_cputs	fclose	fscanf	puts	_spawnvpe
_cscanf	_fcloseall	fseek	raise	system
_debug_calloc	_fdopen	fsetpos	realloc	_tcalloc
_debug_free	feof	ftell	remove	_tempnam
_debug_heapmin	ferror	fwrite	rename	_tfree
_debug_malloc	fflush	_getch	rewind	_tmalloc
_debug_realloc	fgetc	_getche	_rmtmp	tmpfile
_dump_allocated	fgetpos	getenv	scanf	tmpnam
_endthread	fgets	gets	setlocale	_trealloc
_Exception	_fileno	_heap_check	setvbuf	ungetc
_execl	_flushall	_heapmin	signal	_ungetch
_execle	fopen	_kbhit	_spawnl	vfprintf
_execlp	fprintf	_Lib_excpt	_spawnle	vprintf
_execlpe				

OS/2 exceptions in critical functions generally occur only if your program passes a pointer that is not valid to a library function, or if your program overwrites the library's data areas. Because calling a signal handler to handle an OS/2 exception from one of these functions can have unexpected results, a special exception handler is provided for critical functions. **You cannot override this exception handler**.

If the OS/2 exception is synchronous (SIGFPE, SIGILL, or SIGSEGV), the default action is taken, which is to terminate the program and provide a machine-state dump (if the /Tx+ option was specified at compile time). Any exception handler you may have registered will **not** be called, and will receive only the termination exception.

User-Created OS/2 Exception Handlers

If the OS/2 exception is asynchronous, it is deferred until the library function has finished. The exception is then passed to _Exception, which converts the exception to the corresponding C signal and performs the appropriate action.

Note: If you use console I/O functions (for example, gets) and a SIGINT, SIGBREAK, or SIGTERM signal occurs, the signal is deferred until the function returns, for example, after all data for the keyboard function has been entered. To avoid this side effect, use a noncritical function like read or the OS/2 API DosRead to read data from the keyboard.

Creating Your Own OS/2 Exception Handler

You can use OS/2 APIs and the information provided in the Toolkit header file <**bsexcpt.h**> to create your own exception handlers to use alone or with the two provided handler functions. Exception handlers can be complex to write and difficult to debug, but creating your own offers you two advantages:

1. You receive more information about the error condition.
2. You can intercept any OS/2 exception. The C/C++ Tools library passes some exceptions back to the operating system because there is no C semantic for handling them.

Prototype of an OS/2 Exception Handler

The prototype for all exception handlers is:

```
#define INCL_BASE
#include <os2.h>

APIRET APIENTRY MyExceptHandler(EXCEPTIONREPORTRECORD ,
                                EXCEPTIONREGISTRATIONRECORD ,
                                CONTEXTRECORD ,
                                PVOID dummy);
```

where:

APIRET
: Specifies the return type of the function. If you return from your exception handler, you must return one of the following two values:

1. XCPT_CONTINUE_SEARCH indicates that the exception has not been handled and tells the operating system to pass the exception to the next exception handler.
2. XCPT_CONTINUE_EXECUTION indicates that the exception condition has been corrected and tells the operating system to resume running the application using the information in the CONTEXTRECORD.

APIENTRY
: Defines the function linkage. The Toolkit header files define APIENTRY as _System linkage. Use the APIENTRY keyword rather than specifying the linkage type yourself. Note that your exception handler must be an external function; it cannot be static.

EXCEPTIONREPORTRECORD
: Points to a structure that contains high-level information about the exception.

EXCEPTIONREGISTRATIONRECORD
: Points to the record that registered the exception handler. The address of the record is always on the stack.

CONTEXTRECORD
: Points to a structure that contains information about the state of the thread at the time of the exception, including the register contents and the state of the floating-point unit and flags. When an exception handler returns XCPT_CONTINUE_EXECUTION, the machine state is reloaded from this structure. You should only modify the contents of this structure if you are sure your exception handler will return XCPT_CONTINUE_EXECUTION.

PVOID
: Is a pointer to void that you must pass back unchanged to the operating system.

The exception handling structures are defined in the Toolkit header file <**bsexcpt.h**>.

Processing Exception Information

When an exception occurs, the operating system provides a considerable amount of information. Much of it concerns the machine state, which is not particularly useful because it is difficult to relate it to the high-level C language constructs. However, the information contained in the EXCEPTIONREPORTRECORD structure can be quite useful.

The EXCEPTIONREPORTRECORD is defined as:

```
struct _EXCEPTIONREPORTRECORD
   {
   ULONG    ExceptionNum;
   ULONG    fHandlerFlags;
   struct   _EXCEPTIONREPORTRECORD  NestedERR;
   PVOID    ExceptionAddress;
   ULONG    cParameters;
   ULONG    ExceptionInfo[EXCEPTION_MAXIMUM_PARAMETERS];
   };
```

The structure fields provide the following information:

ExceptionNum
: The exception number. There are several exceptions that you will only encounter by using an OS/2 exception handler because the C/C++ Tools default handler passes them to the operating system to handle. They are:

XCPT_PROCESS_TERMINATE
: Indicates that the current thread has called DosExit, and the process is about to end. Until your exception handler ends, the thread continues as though DosExit had not been called.

XCPT_ASYNC_PROCESS_TERMINATE
: Indicates that some other thread in the process has called DosExit and that your current thread is about to end also. You can decide to continue running the current thread and return the exception as handled.

XCPT_ACCESS_VIOLATION
: Indicates an invalid attempt was made to access memory (similar to the SIGSEGV signal). When this exception occurs, the ExceptionInfo field provides the address that generated the exception and the type of access that was attempted (read or write).

XCPT_GUARD_PAGE_VIOLATION
: Indicates that the current thread tried to access a memory page marked as a guard page. Usually it means that your application has accessed a guard page on the stack. In most cases, you will probably pass the exception to the operating system, which will allocate another 4K of committed memory for your thread and a new guard page. The operating system requires about 1.5K to place the information about the exception on the stack and then call the exception handler. If you know you are running out of stack space, you may want to end your process.

XCPT_UNABLE_TO_GROW_STACK
: Indicates that the operating system tried to move your guard page, but no memory remained on the stack. If you suppressed stack probe generation when you compiled (with the /Gs+ option), there may not be enough stack for you to even receive the exception, in which case your process terminates with an operating system trap.

You can also use the DosRaiseException API to create and raise your own exceptions that you can then handle with your own exception handler.

fHandlerFlags
: This field indicates how the exception occurred and what you can do to handle it. It includes the following bits:

EH_NONCONTINUABLE
: You cannot continue running the thread once you leave the exception handler. If you try to return XCPT_CONTINUE_EXECUTION, an error is generated. You cannot reset the bit. However, you can intentionally set the bit to make an exception noncontinuable.

EH_UNWINDING
: A longjmp has been done over this exception handler and the handler is to be deregistered. If your function uses a mutex semaphore (described in the Toolkit documentation), you should release it when you receive this exception.

EH_EXIT_UNWIND
: A DosExit call has been made and the exception has been passed back to the operating system. This exception gives you an opportunity to do something before your exception handler is deregistered.

EH_NESTED_CALL
: An exception occurred while another exception was being handled. This situation should be handled carefully: because each exception requires about 1.5K of stack, nesting exceptions too deep can cause you to run out of stack.

_EXCEPTIONREPORTRECORD NestedERR
: If a nested exception occurs, the information about the exception is found in this structure.

ExceptionAddress
: This field contains the instruction address where the exception occurred. Typically, you cannot determine at run time which function caused the problem.

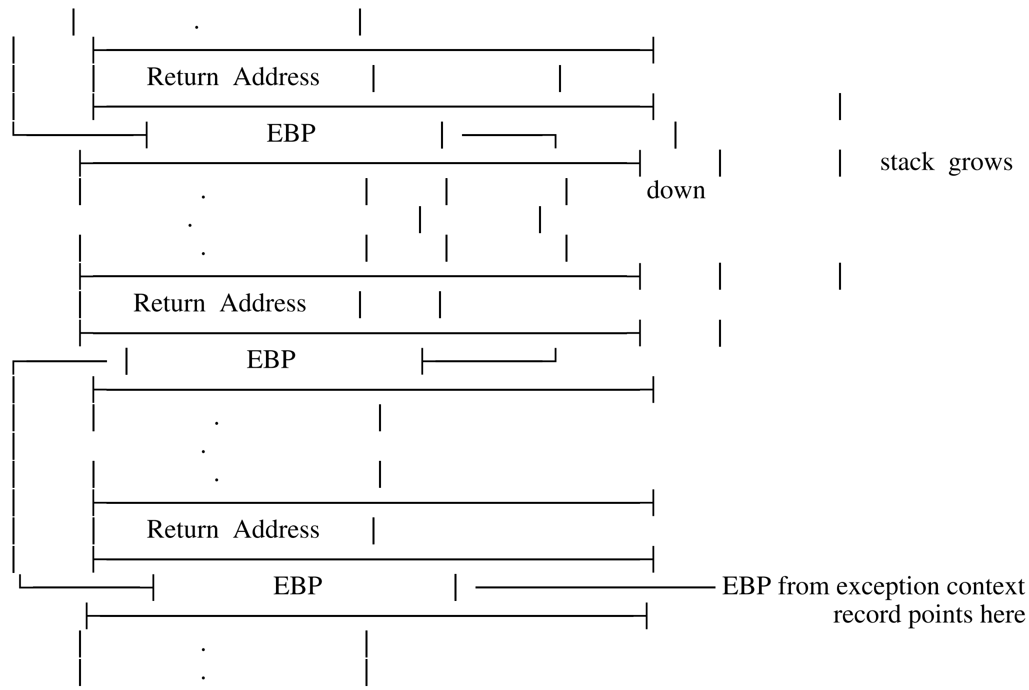
ExceptionInfo
: For some exceptions, this field may contain additional information. For example, if XCPT_ACCESS_VIOLATION occurs, it contains the address at which the memory access failed.

cParameters
: This field contains the number of bytes of information.

The CONTEXTRECORD structure contains information about the machine state of the thread. It is generally of limited use to a high-level programmer because to continue a process after a synchronous exception, you would need to modify the CONTEXTRECORD, and it is extremely difficult to ensure the exception handler code is correct for all possible conditions. You should modify the CONTEXTRECORD only if you have no other alternative to correct your program.

User-Created OS/2 Exception Handlers

You **can** use the CONTEXTRECORD to trace the stack and produce useful debugging information. Because the C/C++ Tools and operating system calling conventions preserve some registers across calls, you cannot reconstruct the registers by traversing the stack to recover from the exception. The 32-bit stack always looks like the following:



Note: If the stack is damaged, you may not be able to trace the EBP chain correctly. You cannot trace over 16-bit calls.

Example of Exception Handling

The following example shows a program similar to the one used for the signal handling example on page 323. In this example, an exception handler is used instead of a signal handler to detect access to memory that is not valid.

```
#define   INCL_DOS
#define   INCL_NOPMAPI
#include <os2.h>
#include <stdlib.h>
#include <setjmp.h>
#include <stdio.h>
#include <stddef.h>            /  for _threadid  /

void   tss_array[1  ];        /  array for 1   thread-specific pointers  /

APIRET APIENTRY MyExceptionHandler(EXCEPTIONREPORTRECORD  ,
                                   EXCEPTIONREGISTRATIONRECORD   ,
                                   CONTEXTRECORD   ,
                                   PVOID);
#pragma map(_Exception,"MyExceptionHandler")
#pragma handler(chkptr)

int chkptr(void   ptr, int size)
{
     volatile char c;          /  volatile to insure access occurs  /
     int valid =  ;            /  count of valid bytes  /
     char   p = ptr;           /  to satisfy the type checking for p++  /
     jmp_buf jbuf;                /  put the jump buffer in automatic storage  /
                                  /      so it is unique to this thread             /
     PTIB ptib;                /  to get the TIB pointer  /
     PPIB  ppib;
     PVOID   temp;
     unsigned int tid =  _threadid;     /  get the thread id  /

     /  create a thread specific jmp_buf  /
     tss_array[tid] = (void  ) jbuf;

     if (!setjmp(jbuf)) {     /  provide a point to return to  /
```

Figure 26 (Part 1 of 3). Example Illustrating an Exception Handler

```
         while (size--)          /  scan the storage  /
         {
              c =  p++;
             valid++;
         }
     }

     ptib->tib_arbpointer = temp;        /  restore the user pointer  /
     return valid;                           /  return number of valid bytes  /
}

/  the exception handler itself  /

APIRET APIENTRY MyExceptionHandler(EXCEPTIONREPORTRECORD   report_rec,
                                   EXCEPTIONREGISTRATIONRECORD   register_rec,
                                   CONTEXTRECORD   context_rec,
                                   PVOID  dummy)
{
    unsigned int tid =  _threadid;      /  get the thread id  /

    /  check the exception flags  /               1
    if (EH_EXIT_UNWIND & report_rec->fHandlerFlags) /  exiting  /
        return  XCPT_CONTINUE_SEARCH;

    if (EH_UNWINDING & report_rec->fHandlerFlags)      /  unwinding  /
        return  XCPT_CONTINUE_SEARCH;

    if (EH_NESTED_CALL & report_rec->fHandlerFlags) /  nested exceptions  /
        return  XCPT_CONTINUE_SEARCH;

    /  determine what the exception is  /     2
    if (report_rec->ExceptionNum == XCPT_ACCESS_VIOLATION) {
         /  this is the one that is expected  /
```

Figure 26 (Part 2 of 3). Example Illustrating an Exception Handler

```
        printf("Detected invalid storage address\n");
        longjmp((int  )tss_array[tid],1);     /  return to the point of the  /
                                                  /  setjmp call without            /
                                                  /  restarting the while loop     /
    } /  endif  /
                                                  3
    return XCPT_CONTINUE_SEARCH;          /  if it is a different exception  /
}
```

Figure 26 (Part 3 of 3). Example Illustrating an Exception Handler

1 The first thing an exception handler should do is check the exception flags. If EH_EXIT_UNWIND is set, meaning the thread is ending, the handler tells the operating system to pass the exception to the next exception handler. It does the same if the EH_UNWINDING flag is set, the flag that indicates this exception handler is being removed.

The EH_NESTED_CALL flag indicates if the exception occurred within an exception handler. If the handler does not check this flag, recursive exceptions could occur until there is no stack remaining.

2 The handler checks the exception number. In general, you should check for only the exceptions that you expect to encounter to protect yourself against possible new exception numbers. Assuming the exception is XCPT_ACCESS_VIOLATION, the exception handler prints a message and calls longjmp to return to the chkptr function.

3 If the exception is not the expected one, the handler tells the operating system to pass it to the next exception handler.

Important: Return XCPT_CONTINUE_EXECUTION from an exception handler **only** if you know that the thread can continue to run because either:

1. The exception is asynchronous and can be restarted.
2. You have changed the thread state so that the thread can continue.

If you return XCPT_CONTINUE_EXECUTION when neither of these conditions is true, you could generate a new exception each time your exception handler ends, eventually causing your process to lock.

Registering an OS/2 Exception Handler

The C/C++ Tools compiler automatically registers and deregisters the _Exception handler for each thread or process so the _Exception is the first exception handler to be called when an exception occurs. To explicitly register _Exception for a function, use the #pragma handler directive before the function definition. This directive generates the code to register the exception handler before the function runs. Code to remove the exception handler when the function ends is also generated.

The format of the directive is:

#pragma handler(*function*)

where *function* is the name of the function or process for which the exception handler is to be registered.

Note: If you use DosCreateThread to create a new thread, you **must** use #pragma handler to register the C/C++ Tools exception handler for the function that the new thread will run.

Registering an OS/2 Exception Handler

You can register your own exception handler in place of _Exception using these directives:

```
#pragma map(_Exception, "MyExceptHandler")
#pragma handler(myfunc)
```

The #pragma map directive tells the compiler that all references to the name _Exception are to be converted to MyExceptHandler. The #pragma handler directive would normally register the exception handler _Exception for the function myfunc, but because of the name mapping, MyExceptHandler is actually registered. The compiler also generates code to deregister MyExceptHandler when myfunc returns.

If you use the method described above, you can have only one exception handler per module. You may need to place functions in separate modules to get the exception handling you want. The handler is registered on function entry and deregistered on exit; you cannot register the handler over only part of a function. For more flexibility, you can use OS/2 APIs to register your exception handler.

The operating system finds exception handlers by following a chain rooted in the thread information block (TIB). When you register an exception handler, you place the address of the handler and the chain pointer from the TIB in an EXCEPTIONREGISTRATIONRECORD structure, and then update the TIB to point to the new EXCEPTIONREGISTRATIONRECORD.

Registering an OS/2 Exception Handler

When you use #pragma handler, the EXCEPTIONREGISTRATIONRECORD is generated and attached to the chain for you. You can register your own records using the DosSetExceptionHandler and DosUnsetExceptionHandler APIs, as shown in the following example:

```
#define INCL_BASE
#include <os2.h>

/  the prototype for the exception handler  /
APIRET APIENTRY MyExceptionHandler(EXCEPTIONREPORTRECORD  ,
                                   EXCEPTIONREGISTRATIONRECORD  ,
                                   CONTEXTRECORD  ,
                                   PVOID);
int myfunction(...)
{
   EXCEPTIONREGISTRATIONRECORD err = { NULL,MyExceptionHandler };

   DosSetExceptionHandler(&err);   /  register  /
  .
  .
  .
   DosUnsetExceptionHandler(&err);   /  deregister  /
}
```

Using the OS/2 APIs provides more flexibility than using #pragma handler. You can register the exception handler over only a part of the function if you want. You can also register more than one exception handler for a function. When you use DosSetExceptionHandler to register your handler, you can also make the EXCEPTIONREGISTRATIONRECORD part of a larger structure and then access the information in that structure from inside the exception handler.

You **must** deregister the exception handler before the function ends. If you do not, the next exception that occurs on the thread can have unexpected and undefined results. When you use #pragma handler, the exception handler is automatically deregistered for you.

The following diagram shows the TIB chain:

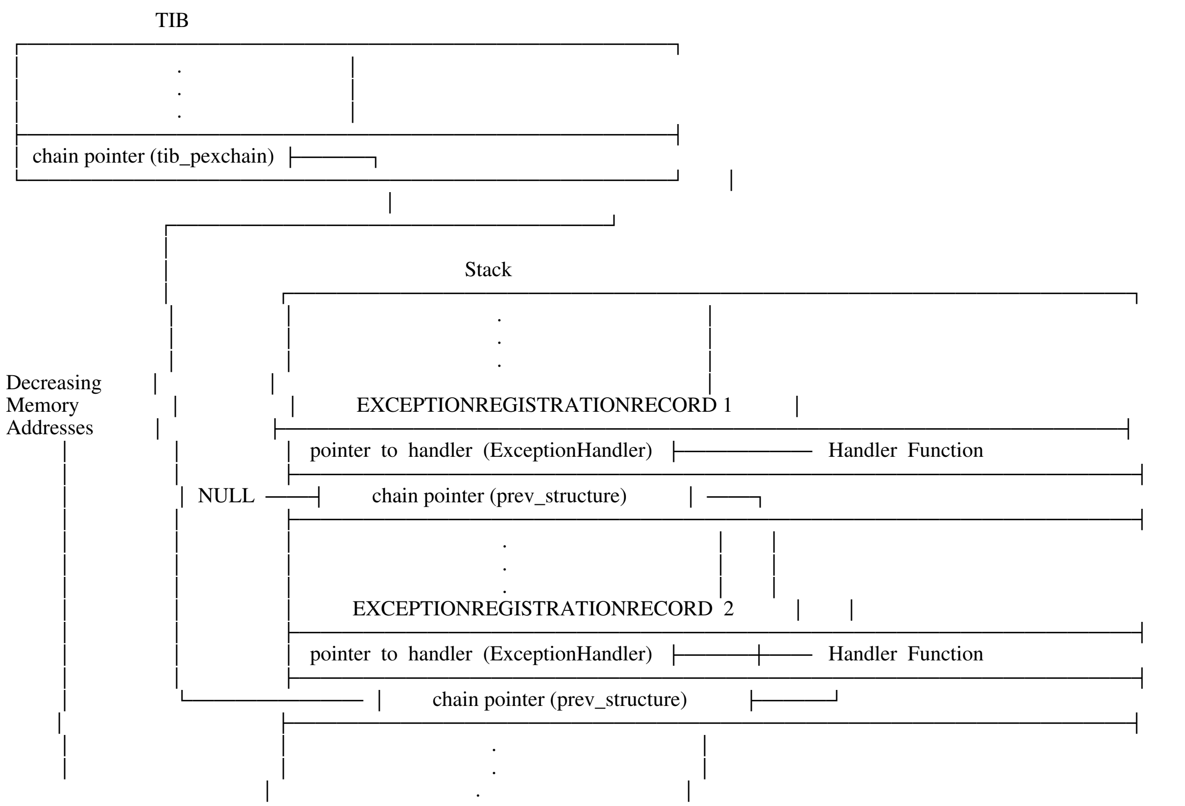


Figure 27. TIB Chain. Names in parentheses are the names of the fields of the EXCEPTIONREGISTRATIONRECORD *structure.*

Each EXCEPTIONREGISTRATIONRECORD is chained to the next. When an exception occurs, the operating system begins at the TIB and goes to each EXCEPTIONREGISTRATIONRECORD in turn. It calls the exception handler and passes it the exception information. The exception handler either handles the exception or tells the operating system to pass the exception to the next handler in the chain. If the last exception handler in the chain, identified by the NULL chain pointer, does not handle the exception, the operating system takes the default action.

An EXCEPTIONREGISTRATIONRECORD must be on the stack, and each record must be at a higher address than the previous one.

Handling Signals and OS/2 Exceptions in DLLs

Handling signals and OS/2 exceptions in DLLs is no different than handling signals in executable files, providing all your DLLs and the executable files that use them are created using the C/C++ Tools compiler, and only one C/C++ Tools library environment exists for your entire application (your executable module and all DLLs).

The library environment is a section of information associated with and statically linked to the C/C++ Tools library itself. You can be sure your program has only one library environment if:

1. It consists of a single executable module. By definition, a single module has only one copy of the C/C++ Tools library environment regardless of whether it links to the library statically or dynamically.
2. Your executable module dynamically links to a single DLL that is statically bound to the C/C++ Tools runtime library and that uses the C/C++ Tools library functions. The executable module then accesses the library functions through the DLL.
3. Your executable modules and DLLs all dynamically link to the C/C++ Tools runtime library.

Note: The licensing agreement does not allow you to ship the C/C++ Tools library DLLs with your application. You can, however, create your own version of the runtime library and dynamically link to it from all of your modules, ensuring that only one copy of the library environment is used by your application. If you call any C/C++ Tools library functions from a user DLL, you must call them all from that DLL. The method of creating your own runtime library is described in "Creating Your Own Runtime Library DLLs" on page 216.

If more than one of your modules is statically linked to the C/C++ Tools library, your program has more than one library environment. Because there is no communication between these environments, certain operations and functions become restricted:

- Stream I/O. You can pass the file pointer between modules and read to or write from the stream in any module, but you cannot open a stream in one library environment or module and close it in another.
- Memory allocation. You can pass the storage pointer between modules, but you cannot allocate storage in one library environment and free or reallocate it in another.
- strtok, rand, and srand functions. A call to any of these functions in one library environment has no effect on calls made in another environment.
- errno and _doserrno values. The setting of these variables in one library environment has no effect on their values in another.
- Signal and OS/2 exception handlers. The signal and exception handlers for a library environment have no effect on the handlers for another environment.

In general, it is easier to use only one library environment, but not always possible. For example, if you are building a DLL that will be called by a number of applications, you should assume that there may be multiple library environments and code your DLL accordingly.

The following section describes how to use signal and exception handling when your program has more than one library environment.

Signal and Exception Handling with Multiple Library Environments

When you have multiple library environments, you must treat signal and exception handlers in a slightly different manner than you would with a single library environment. Otherwise, the wrong handler could be called to handle a signal or OS/2 exception.

For example, if you have an executable module and a DLL, each with its own library environment, the _Exception exception handler is automatically registered for the executable module when it starts. When the executable module calls a function in the DLL, the thread of execution passes to the DLL. If an OS/2 exception then occurs in the code in the DLL, it is actually handled by the exception handler in the executable module's library environment. Any signal handling set up in the DLL is ignored.

When you have more than one library environment, you must ensure that an OS/2 exception is always handled by the exception handler for the library environment where the exception occurred.

Include #pragma handler statements in your DLL for every function in the DLL that can be called from another module. This directive ensures the exception handler for the DLL's library environment is correctly registered when the function is called and deregistered when the function returns to the calling module. If functions in your executable module can themselves be called back to from a DLL, include a #pragma handler statement for each of them also.

Using OS/2 Exception Handlers for Special Situations

Using exception handlers can be especially helpful in the following situations:

- In multithread programs that use OS/2 semaphores. If you acquire a semaphore and then use longjmp either explicitly or through a signal handler to move to another place in your program, the semaphore is still owned by your code. Other threads in your program may not be able to obtain ownership of the semaphore.

  If you register an exception handler for the function where the semaphore is requested, the handler can check for the unwind operation that occurs as a result of a longjmp call. If it encounters an unwind operation, it can then release the semaphore.

- In system DLLs. Using an exception handler allows you to run process termination routines even if your DLL has global initialization and termination.

  When a process terminates, functions are called in the following order:

  1. Functions registered with the atexit or _onexit functions.
  2. Exception handlers for termination exceptions.
  3. Functions registered with the DosExitList API.
  4. DLL termination routines.

  You can include process termination routines in your exception handler and they will be performed before the DLL termination routines are called.

OS/2 Exception Handling Considerations

All the restrictions for signal handling described on page 326 apply to exception handling as well. There are also a number of additional considerations you should keep in mind when you use exception handling:

- You **must** register an exception handler whenever you change library environments to ensure that exception handling is provided for all C code.
- If you register your own exception handler, the OS/2 exceptions you handle are not seen by a signal handler. The exceptions you do not handle are passed to the next exception handler. If the next handler is the C/C++ Tools default handler _Exception, it converts the exception to a signal and calls the appropriate signal handler.
- Ensure that you always deregister your exception handler. If you do not, your process typically ends abnormally. It is very difficult to discover this problem through debugging. If you use #pragma handler, the handler is automatically deregistered; if you use the OS/2 APIs, you must call DosUnsetExceptionHandler.
- If you are using OS/2 semaphores and an exception occurs while your code owns a semaphore, you must ensure that the semaphore is released. You can release the semaphore either by continuing the exception or by explicitly releasing the semaphore in the signal handler.
- Always check the exception flags to determine how the exception occurred. Any exception handler can be unwound by a subsequent handler.
- Keep your exception handler simple and specific. Exception handlers are easier to write and maintain if you limit what they can do. A handler that does everything can be very large and very complicated.
- Check for and handle only the exceptions that you expect to encounter, and provide a default exception handler to handle the unexpected. If the operating system adds new exceptions, or if you create your own, the default handler will handle them.

If you are using your own exception handler, it receives the exception registration record when an exception occurs, as described in “Registering an OS/2 Exception Handler” on page 344. Do **not** use the return address of the calling function to tell you where to resume execution, because the values of the registers other than EBP (for example, EBX, EBI, and EDI) at the return are generally not available to your exception handler.

You need approximately 1.5K of stack remaining for the operating system to be able to call your exception handler. If you do not have enough stack left, the operating system terminates your process.

Neither of the C/C++ Tools default exception handlers are available in the subsystem libraries. Because the subsystem libraries contain no critical or math functions, the _Lib_excpt function is not required.

Restricted OS/2 APIs

When you use the C/C++ Tools default exception handlers, certain OS/2 APIs can interfere with exception handling:

DosCreateThread
: This API does not automatically register an exception handler for the new thread. Use _beginthread instead, or use #pragma handler before the DosCreateThread call to register the handler for the thread.

DosExit
: This API does not perform all necessary library termination routines. Instead, use exit or _exit, abort, or _endthread, or simply fall out of main.

DosUnwindException
: This API can unwind the C/C++ Tools exception handlers from the stack. Use longjmp instead.

DosSetSignalExceptionFocus
: Using this API to remove the signal focus from a C/C++ Tools application may prevent you from receiving SIGINT and SIGBREAK exceptions from the keyboard.

DosAcknowledgeSignalException
: This API interferes with the C/C++ Tools handling of signal exceptions. The library exception handler acknowledges signals for you.

DosEnterMustComplete
: This API can be used to delay the handling of asynchronous exceptions, including termination exceptions, until a section of code has ended. You must call DosExitMustComplete at the end of the section to reenable the exception handling.

DosEnterCritSec
: This API prevents execution from switching between threads until a section of code has ended. You must call DosExitCritSec at the end of the critical section of code. Use these APIs only if you cannot use a mutex semaphore. If you must use them, keep critical sections short and avoid including calls that may get blocked.

Handling Floating-Point Exceptions

Floating-point exceptions require special exception handling. In general, you cannot retry a floating-point exception without a significant knowledge of both the 80387 chip and the application that generated the exception. Because knowledge of your application is beyond the capabilities of the C/C++ Tools library, it treats a floating-point exception as a terminating condition.

You can use the _control87 function and the bit mask values defined in <**float.h**> to mask floating-point exceptions, that is, to prevent them from being reported. Each bit mask corresponds to a unique floating-point exception that can be masked individually. Masking exceptions also changes the state of the floating-point control word for the 80387 chip. When a floating-point exception is masked, the 80387 chip performs a predetermined corrective action.

The bit masks are:

EM_INVALID
: Mask exceptions resulting from floating-point operations that are not valid. Such an exception can be caused by a floating-point value that is not valid, such as a signalling NaN, or by a problem with the 80387 stack. The corrective action taken by the 80387 chip is to return a quiet NaN.

  Note: Because this type of exception indicates a serious problem, you should not mask it off.

EM_DENORMAL
: Mask exceptions resulting from the use of denormal floating-point values. The corrective action is to use these values and allow for gradual underflow. This type of exception is not meaningful under the C/C++ Tools compiler and is masked off by default.

EM_ZERODIVIDE
: Mask the divide-by-zero exception. The 80387 chip returns a value of infinity.

EM_OVERFLOW
: Mask the overflow exception. The 80387 chip returns a value of infinity.

EM_UNDERFLOW
: Mask the underflow exception. The 80387 chip returns either a denormal number or zero.

EM_INEXACT
: Mask the exception that indicates precision has been lost. Because this type of exception is only useful when performing integer arithmetic, while the 80387 chip is used for floating-point arithmetic only, the exception is not meaningful and the 80387 chip ignores it. This exception is masked off by default.

For example, to mask the floating-point underflow exception from being reported, you would code in your source file:

```
oldstate = _control87(EM_UNDERFLOW, EM_UNDERFLOW);
```

To mask it on again, you would code:

```
oldstate = _control87( , EM_UNDERFLOW);
```

You can also reset the entire floating-point control word to the default state with the _fpreset function. Both _fpreset and _control87 are described in the *C Library Reference*.

Important: Because the C/C++ Tools math functions defined in <**math.h**> use the 80387 chip, make sure that when you call any of them, the floating-point control word is set to the default state to ensure exceptions are handled correctly by the C/C++ Tools library.

Note also that the state of the floating-point control word is unique for each thread, and changing it in one thread does not affect any other thread.

Interpreting Machine-State Dumps

Note: This section provides information to be used for Diagnosis, Modification, or Tuning purposes. This information is **not** intended for use as a programming interface.

If you specify the /Tx+ option, when a process is ended because of an unhandled or incorrectly handled exception, the exception handler performs a machine-state dump. A machine-state dump consists of a number of runtime messages that show information about the state of the system, such as the contents of the registers and the reason for the exception. This information is sent to file handle 2, which is usually associated with stderr. You can also use the _set_crt_msg_handle function to redirect the messages to a file. See the *C Library Reference* for more information about this function.

If you do not specify /Tx+, a message is generated giving the exception and the address at which it occurred.

For example, the following program generates a floating-point exception. Because the exception cannot be handled, a machine-state dump is performed. Figure 29 on page 358 shows what is sent to stderr and explains the messages in the dump.

```
#include <math.h>

int main(void)
{
   _Packed union SIGNAN {           / a union which allows us to set /
      double dbl;                   /   the parts of a double value   /
      _Packed struct {
         unsigned int siglow : 26;
         unsigned int sighigh : 26;
         unsigned int exp : 11;
         unsigned int sign : 1;
      } dblrep;
   } signan;
   double x;

   / set the double value to a signalling /
   / NaN, which the library cannot handle /
   signan.dblrep.sign = ;
   signan.dblrep.exp = x7ff;
   signan.dblrep.sighigh = ;
   signan.dblrep.siglow = 1;

   / now call a math function with a /
   /   signalling NaN to cause a trap /
   x = atan2(signan.dbl,2. );

   / the program never gets here /
   return ;
}
```

Figure 28. Program to Cause a Machine-State Dump

```
Floating Point Invalid Operation exception occurred at EIP =    5     on
 thread   1.           1
Exception occurred in C Library routine called from EIP =    112D8.  2
Register Dump at point of exception:    3
EAX =       1       EBX =            ECX =    B  1       EDX =   14  1
EBP =                EDI =           ESI =    61FCC      ESP =    61FA8    4
 CS =           5B    CSLIM = 1BFFFFFF     DS =          53    DSLIM = 1BFFFFFF
 ES =           53    ESLIM = 1BFFFFFF     FS =         15 B    FSLIM =      3
 GS =                GSLIM =             SS =          53    SSLIM = 1BFFFFFF
NPX Environment:    5
CW =  362          TW = 3FFF    IP =   5B:   1  2B      6
SW = B881    OPCODE =  545    OP =   53:   23414
NPX Stack:    7
ST(7): exponent =      significand = +                    8
Process terminating.      9
```

Figure 29. Example of a Machine-State Dump

1 The first line always states the nature of the exception and the place and thread where the exception occurred. If you specify /Tx-, this is the only message that is generated.

2 Indicates that the exception occurred within one of the C library functions. It also indicates the place and thread where the call to that library function was made.

You can use the address given in 1 and 2 to determine where in your code the problem occurred. To do this, you must create a map file by specifying either the compiler option /B"/map", or if you are linking your program separately, the linker option /map.

3 Introduces the register dump.

4 Gives the values contained by each register at the time the exception occurred. For information on the purpose of each register, see the documentation for your processor chip.

5 Introduces the state of the numeric processor extension (NPX) at the time of the exception.

6 Gives the values of the elements in the NPX environment.

7 Introduces the state of the NPX stack at the time of the exception.

8 One copy of this message appears for each valid stack entry in the NPX and gives the values for each. In this example, because there is only one stack entry, the message appears only once. If there are no valid stack entries, a different message is issued in place of this message to state that fact.

9 Confirms that the process is terminating. It is one of several informational messages that may accompany the initial exception message and register dump.

In general, a dump will always include items 1 , 3 , and 4 . Item 2 appears only if the exception occurred in a C/C++ Tools library function. Items 5 to 8 appear only if the NPX was in use at the time of the exception. Item 9 may or may not appear, depending on the circumstances of each exception.

For a list of all the runtime messages and their explanations, see the *Online Language Reference*.

Note: If you copy and run the program in Figure 28 on page 357, you will get the same messages as shown in Figure 29 on page 358, but the values given may be different.

Machine-State Dumps

Part 6. Appendixes

Appendixes

Appendix A. ANSI Notes on Implementation-Defined Behavior

The C/C++ Tools product supports the requirements of the *American National Standard for Information Systems / International Standards Organization – Programming Language C* standard, ANSI/ISO 9899-1990[1992], and the current draft of the *Working Paper for Draft Proposed American National Standard for Information Systems - Programming Language C++* ANSI X3J16/92-00091, (September 17, 1992), as understood and interpreted by IBM as of March 1993. It also supports the IBM SAA C standards as documented in the *SAA CPI C Reference - Level 2*. This appendix describes how the C/C++ Tools product behaves where the ANSI C Standard describes behavior as implementation-defined. These behaviors can affect your writing of portable code.

Implementation-Defined Behavior Common to Both C and C++

The following sections describe how the C/C++ Tools product defines the behavior classified as implementation-defined in the ANSI C Standard.

Identifiers

The number of significant characters in an identifier with no external linkage is 255.

The number of significant characters in an identifier with external linkage is 255.

The C/C++ Tools compiler truncates all external names to 255 characters.

Case sensitivity: When you perform the compile and link steps separately, the case of identifiers is ignored unless you specify the /NOIGNORECASE (/NOI) linker option. If you use the icc command to invoke the linker, the /NOI option is automatically supplied for you, and the case of identifiers is significant.

Characters

A character is represented by 8 bits, as defined by the CHAR_BIT macro in <**limits.h**>.

The same code page is used for the source and execution set. (Source characters and strings do not need to be mapped to the execution character set.)

When an integer character constant contains a character or escape sequence that is not represented in the basic execution character set, the char is assigned the character after the backslash, and a warning is issued. For example, '\q' is interpreted as the character 'q'.

When a wide character constant contains a character or escape sequence that is not represented in the extended execution character set, the wchar_t is assigned the character after the backslash, and a warning is issued.

When an integer character constant contains more than one character, the last 4 bytes represent the character constant.

When a wide character constant contains more than one multibyte character, the last wchar_t value represents the character constant.

The default behavior for char is unsigned.

Any sequential spaces in your source program are interpreted as one space.

All spaces are retained for the listing file.

Strings

The C/C++ Tools compiler provides the following additional sequence forms for strtod, strtol, and strtoul functions in locales other than the C locale:

inf   infinity   nan

All of these sequences are not case sensitive.

When you use DBCS (with the /Sn compiler option), a hexadecimal character that is a valid first byte of a double-byte character is treated as a double-byte character inside a string. That is, a is appended to the character that ends the string. Double-byte characters in strings **must** appear in pairs.

Integers

Figure 30. Integer Storage and Range

Type	Amount of Storage	Range (in <limits.h>)
signed short	2 bytes	-32768 to 32767
unsigned short	2 bytes	to 65535
signed int	4 bytes	-2147483648 to 2147483647
unsigned int	4 bytes	to 4294967295
signed long	4 bytes	-2147483648 to 2147483647
unsigned long	4 bytes	to 4294967295

Note: Do not use the values in this table as numbers in a source program. Use the macros defined in <**limits.h**> to represent these values.

When you convert an integer to a signed char, the least-significant byte of the integer represents the char.

When you convert an integer to a short signed integer, the least-significant 2 bytes of the integer represents the short int.

When you convert an unsigned integer to a signed integer of equal length, if the value cannot be represented, the magnitude is preserved and the sign is not.

When bitwise operations (OR, AND, XOR) are performed on a signed int, the representation is treated as a bit pattern.

The remainder of integer division is negative if exactly one operand is negative.

When either operand of the divide operator is negative, the result is truncated to the integer value and the sign will be negative.

ANSI Notes

The result of a bitwise right shift of a negative signed integral type is sign extended.

The result of a bitwise right shift of a non-negative signed integral type or an unsigned integral type is the same as the type of the left operand.

Floating-Point Values

Figure 31. Floating Point

Type	Amount of Storage	Range of Exponents (base 10) (in <float.h>)
float (IEEE 32-bit)	4 bytes	-37 to 38
double (IEEE 64-bit)	8 bytes	-3 7 to 3 8
long double (IEEE 80-bit)	16 bytes	-4931 to 4932

When an integral number is converted to a floating-point number that cannot exactly represent the original value, it is truncated to the nearest representable value.

When a floating-point number is converted to a narrower floating-point number, it is rounded to the nearest representable value.

Arrays and Pointers

The type of the integer required to hold the maximum size of an array (the type of the sizeof operator, size_t) is unsigned int.

The type of the integer required to hold the difference between two pointers to elements of the same array (ptrdiff_t) is int.

When you cast a pointer to an integer or an integer to a pointer, the bit patterns are preserved.

Registers

The C/C++ Tools compiler optimizes register use and does not respect the register storage class specifier.

In C programs, you cannot take the address of an object with a register storage class. This restriction does not apply to C++ programs.

Structures, Unions, Enumerations, Bit-Fields

If a member of a union object is accessed using a member of a different type, the result is undefined.

If a structure is not packed, padding is added to align the structure members on their natural boundaries and to end the structure on its natural boundary. The alignment of the structure or union is that of its strictest member. If the length of the structure is greater than a doubleword, the structure is doubleword-aligned. The alignment of the individual members is not changed. Packed structures are not padded. See Appendix C, "Mapping" on page 385 for more information.

The default type of an integer bit field is unsigned int.

Bit fields are allocated from low memory to high memory, and the bytes are reversed. For more information, see Appendix C, "Mapping" on page 385.

Bit fields can cross storage unit boundaries.

The maximum bit field length is 32 bits. If a series of bit fields does not add up to the size of an int, padding may take place.

A bit field cannot have type long double.

The expression that defines the value of an enumeration constant cannot have type long double.

An enumeration can have the type char, short, or long and be either signed or unsigned, depending on its smallest and largest values.

In C++, enumerations are a distinct type, and although they may be the same size as a data type such as char, they are not considered to be of that type.

Qualifiers

All access to an object that has a type that is qualified as volatile is retained.

Declarators

There is no C/C++ Tools limit for the maximum number of declarators (pointer, array, function) that can modify an arithmetic, structure, or union type. The only constraint is your system resources.

Statements

Because the case values must be integers and cannot be duplicated, the maximum number of case values in a switch statement is 4 294 967 296.

Preprocessor Directives

The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the character constant in the execution character set.

Such a constant can have a negative value.

For the method of searching system include source files (< >), see "Controlling #include Search Paths" on page 38.

User include files can be specified in double quotation marks (" ... "), see "Controlling #include Search Paths" on page 38.

For the mapping between the name specified in the include directive and the external source file name, see "Controlling #include Search Paths" on page 38.

For the behavior of each #pragma directive, see the *Online Language Reference* or the *C Language Reference* and *C++ Language Reference*.

The __DATE__ and __TIME__ macros are always defined as the system date and time.

Library Functions

In extended mode (the default) and for all C++ programs, the NULL macro is defined to be . For all other language levels, NULL is defined to be
((void)).

When assert is executed, if the expression is false, the diagnostic message written by the assert macro has the format:

Assertion failed: *expression*, file *file_name*, line *line_number*

To create a table of the characters set up by the CTYPE functions, use the program in Figure 32 on page 370. The columns are organized by function as follows:

(Column 1)	The hexadecimal value of the character
AN	isalnum
A	isalpha
C	iscntrl
D	isdigit
G	isgraph
L	islower
(Column 8)	isprint
PU	ispunct
S	isspace
PR	isprint
U	isupper
X	isxdigit

The value returned by all math functions after a domain error (EDOM) is a NaN.

The value errno is set to on underflow range errors is ERANGE.

If you call the fmod function with as the second argument, fmod returns and a domain error.

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

int main(void)
{
    int  ch;

     for (ch =  ; ch <=  xff; ch++)
         {
          printf("%# 4X ", ch);
          printf("%3s ", isalnum(ch)    ? "AN" : " ");
          printf("%2s ", isalpha(ch)    ? "A"    : " ");
          printf("%2s",   iscntrl(ch)   ?  "C"   :  "  ");
          printf("%2s",   isdigit(ch)   ?  "D"   :  "  ");
          printf("%2s",   isgraph(ch)   ?  "G"   :  "  ");
          printf("%2s",   islower(ch)   ?  "L"   :  "  ");
          printf("%c",   isprint(ch)   ?  ch      :  '  ');
          printf("%3s",   ispunct(ch)   ?  "PU"  :  "  ");
          printf("%2s",   isspace(ch)   ?  "S"   :  "  ");
          printf("%3s",   isprint(ch)   ?  "PR"  :  "  ");
          printf("%2s",   isupper(ch)   ?  "U"   :  "  ");
          printf("%2s",    isxdigit(ch) ? "X"    : " ");

         putchar('\n');
         }
         return   ;
}
```

Figure 32. C Program to Print out CTYPE Characters

Error Handling

See the *Online Language Reference* for a list of the runtime messages generated for perror and strerror. Note that the value of errno is not generated with the message.

See the *Online Language Reference* for the lists of the messages provided with the C/C++ Tools compiler.

Messages are classified as shown by the following table:

Type of Message	Return Code
Information	
Warning	
Error	12
Severe error	16 or 2 or 99

Use the /W*n* compile-time option to control the level of messages generated. There is also a /W*grp* compiler option that provides programming-style diagnostics to aid you in determining possible programming errors. See "Debugging and Diagnostic Information Options" on page 92.

Signals

The set of signals for the signal function is described in Chapter 18, "Signal and OS/2 Exception Handling" on page 317.

The parameters and the usage of each signal recognized by the signal function are described in Chapter 18, "Signal and OS/2 Exception Handling" on page 317 and in the *C Library Reference* under signal.

SIG_DFL is the default signal, and the default action taken is termination. See Chapter 18, "Signal and OS/2 Exception Handling" on page 317 for more information on signal handling.

If the equivalent of signal(sig, SIG_DFL); is not executed at the beginning of signal handler, no signal blocking is performed. See Chapter 18, "Signal and OS/2 Exception Handling" on page 317.

Whenever you leave a signal handler, it is reset to SIG_DFL.

Translation Limits

The following table lists the C/C++ Tools translation limits:

Figure 33. Translation Limits

Nesting levels of:	
Compound statements	No limit
Iteration control	No limit
Selection control	No limit
Conditional inclusion	No limit
Parenthesized declarators	No limit
Parenthesized expression	No limit
Number of pointer, array and function declarators modifying an arithmetic, a structure, a union, and incomplete type declaration	No limit
Significant initial characters in:	
Internal identifiers	255
Macro names	No limit
External identifiers	255
Number of:	
External identifiers in a translation unit	1024
Identifiers with block scope in one block	No limit
Macro identifiers simultaneously declared in a translation unit	No limit
Parameters in one function definition	255
Arguments in a function call	255
Parameters in a macro definition	No limit
Parameters in a macro invocation	No limit
Characters in a logical source line	No limit
Characters in a string literal	No limit
Size of an object (in bytes)	LONG_MAX
Nested #include files	127 (C), 256 (C++)
Enumeration constants in an enumeration	4 294 967 296 distinct values
Levels in nested structure or union	No limit

Streams and Files

The last line of a text stream does not require a terminating new-line character.

Space characters that are written out to a text stream immediately before a new-line character appear when read.

When a text stream is connected to a character device, the Ctrl-Z (\x1a) character is treated as an end-of-file indicator.

If Ctrl-Z is the last character in a file, it is treated as the end-of-file. Similarly, when a file ending with a Ctrl-Z character is opened in append mode, the Ctrl-Z is discarded. This Ctrl-Z behavior applies to text mode only.

There is no limit to the number of null characters that can be appended to the end of a binary stream.

The file position indicator of an append mode stream is positioned at the end of the file.

When a file is opened in write mode, the file is truncated. If the file does not exist, it is created.

A file of zero length does exist.

For the rules for composing a valid file name, refer to the documentation for the OS/2 operating system.

For reading, the same file can be simultaneously opened multiple times; for writing or appending, the file can be opened only once.

When the remove function is used on an open file, remove fails.

When you use the rename function to rename a file to a name that exists prior to the function call, rename fails.

Temporary files may not be removed if the program terminates abnormally.

When the tmpnam function is called more than TMP_MAX times, tmpnam fails and returns NULL, and sets errno to ENOGEN.

The output of %p conversion in the fprintf function is equivalent to %x.

The input of %p conversion in the fscanf function is the same as is expected for %x.

ANSI Notes

A '-' character that is neither the first nor the last in the scan list for %[] conversion in the fscanf function is considered to be part of the scan list.

The possible values of errno on failure of fgetpos are EERRSET, ENOSEEK, and EBADPOS.

The possible values of errno on failure of ftell are EERRSET, ENOSEEK, EBADPOS, and ENULLFCB.

Memory Management

If the size requested is , the calloc, malloc, and realloc functions all return a NULL pointer. In the case of realloc, the pointer passed to the function is also freed.

Environment

You can pass arguments to main through argv, argc, and envp.

If a standard stream is redirected to a file, the stream is fully buffered, with the exception of stderr, which is line buffered. If the standard stream is attached to a character device, it is line buffered.

When the abort function is called, all open files are closed by the operating system. The buffers are not flushed. Any memory files belonging to the process are discarded.

When the abort function is called, the return code of 3 is returned to the host environment.

When a program ends successfully and calls the exit function with the argument or EXIT_SUCCESS, all buffers are flushed, all files are closed, all storage is released, and the argument is returned.

When a program ends unsuccessfully and calls the exit function with the argument EXIT_FAILURE, all buffers are flushed, all files are closed, all storage is released, and the argument is returned.

If the argument passed to the exit function is other than , EXIT_FAILURE or EXIT_SUCCESS, all buffers are flushed, all files are closed, all storage is released, and the argument is returned.

For the set of environmental names, see Chapter 7, “Setting Runtime Environment Variables” on page 133 and “OS/2 Environment Variables for Compiling” on page 34.

For the method of altering the environment list obtained by a call to the getenv function, see the _putenv function in the *C Library Reference*.

For the format and mode of execution of a string on a call to the system function, see the *C Library Reference* under system.

Localization

The environment specified by "" locale on a call to setlocale is the C default locale.

The supported locales are listed in Appendix E, “Component Files” on page 431.

Time

The local time zone and daylight saving time zone are EST and EDT. See Chapter 7, “Setting Runtime Environment Variables” on page 133 and the _tzset function in the *C Library Reference* for more information on specifying the time zone.

The era for the clock function starts when the program is started by either a call from the operating system or a call to system.

C++-Specific Implementation-Defined Behavior

The following sections describe how the C/C++ Tools product defines the behavior classified as implementation-defined in the ANSI C++ Working Paper.

Classes, Structures, Unions, Enumerations, Bit Fields

Class members are allocated in the order declared; access specifiers have no effect on the order of allocation.

Padding is added to align class members on their natural boundaries and to end the class on its natural boundary.

An int bit field behaves as an unsigned int for function overloading.

Linkage Specifications

The valid values for the string literal in a linkage specification are:

"C++" Default

"C" C language linkage

Member Access Control

Class members are allocated in the order declared; access specifiers have no effect on the order of allocation.

Special Member Functions

Temporary objects are generated under the following circumstances:

- During reference initialization
- During evaluation of expressions
- In type conversions
- Argument passing
- Function returns
- In throw expressions.

Temporary objects exist until there is a break in the flow of control of the program. They are destroyed on exiting the scope in which the temporary object was created.

Migrating Headers from 16-bit C to 32-bit C/C++

The following section describes the changes to existing 16-bit C headers that are needed in order to k with both 32-bit C and C++ code.

Structures

#pragma pack statements should be added around structure declarations of structures that will be passed to or from 16-bit code. Do not use the _Packed keyword because it is not supported by C++.

Integers declared in the structures should be specifically declared as short or long, not int so that the structures have the same size and layout in both 16-bit and 32-bit code.

Create typedefs for your structures, and use #pragma seg16 on those typedefs to specify that those structures should not cross a 64K boundary when laid out in memory.

Any structure members that are pointers must have the pointer qualified with the _Seg16 type qualifier. For example, far would be translated to _Seg16. This may even need to be done recursively if the 16-bit code will be manipulating the object pointed at.

Function Prototypes

Prototype your functions using the linkage convention keywords. Do not use #pragma linkage because it is not supported in C++.

Any functions that take pointers to other functions should have the linkage of the function pointed at specified in the prototype. This will avoid errors when the /Mp or /Ms compiler options are used to set the default linkage.

For functions that take pointers, if the pointer is passed between 32-bit and 16-bit code as part of an aggregate or class, or uses more than one level of indirection (for example, a pointer to a pointer, you must qualify the pointer with _Seg16. For example, far would be translated to _Seg16. If the pointer is passed directly, the _Seg16 keyword is not required.

Required Conditional Compilation Directives

The following directives must be added to the beginning of each header file:

```
#if __cplusplus
 extern "C" {
#endif
```

The following directives must be added to the end of each header file:

```
#if __cplusplus
}
#endif
```

Migrating Headers from (32-bit) C Set/2 Version 1 to (32-bit) C++

The following changes to your existing header files are needed in order work with both C and C++ code:

Any use of the _Packed keyword must be removed and replaced with the appropriate use of #pragma pack. C++ does not support _Packed.

Any use of #pragma linkage must be removed and the appropriate linkage convention keyword must be added to the prototype. C++ does not support #pragma linkage directives.

The following must be added to the beginning of each header file:

```
#if __cplusplus
 extern "C" {
#endif
```

The following must be added to the end of each header file:

```
#if __cplusplus
}
#endif
```

Creating New Headers to Work with Both C and C++ (32-bit)

The following are needed in your new header files in order work with both C and C++ code:

- The following must be added to the beginning of each header file:

```
#if __cplusplus
 extern "C" {
#endif
```

- The following must be added to the end of each header file:

```
#if __cplusplus
}
#endif
```

- Do not use _Packed in your code; use #pragma pack instead.
- Do not use #pragma linkage in your code; use the linkage convention keywords instead.
- Use typedefs for any structures being passed to 16-bit code and specify the typedef in a #pragma seg16 statement.
- Specify the linkage on any variables that are pointers to functions.
- Use the _Seg16 type qualifier to declare external pointers that will be shared between 32-bit and 16-bit code, that is, that are declared in both. The _Seg16 qualifier directs the compiler to store the pointer as a segmented pointer (with a 16-bit selector and 16-bit offset) that can be used directly by a 16-bit application. You can also use the pointer in a 32-bit program; the C/C++ Tools compiler automatically converts it to a flat 32-bit pointer when necessary.

ANSI Notes

Appendix B. C/C++ Tools Macros and Functions

This appendix lists the predefined macros reserved for use by the C/C++ Tools product. It also includes a list of the intrinsic and built-in functions. For a complete list of all functions in the C/C++ Tools runtime libraries, see the *C Library Reference* or *Reference Summary*.

Predefined Macros

| The macros identified in this section are provided to allow customers to write programs that use C/C++ Tools services. Only those macros identified in this section should be used to request or receive C/C++ Tools services.

The C/C++ Tools compiler provides both the SAA predefined macros and a number of macros specific to the C/C++ Tools product.

Macro	Description
_CHAR_UNSIGNED	Indicates default character type is unsigned. Defined using the #pragma chars directive or /J compiler option.
_CHAR_SIGNED	Indicates default character type is signed. Defined using the #pragma chars directive or /J compiler option.
__COMPAT__	Indicates language constructs compatible with earlier versions of the C++ language are allowed. Defined using the #pragma langlvl(compat) directive or /Sc compiler option. This macro is valid for C++ programs only.
__cplusplus	Set to the integer 1. Indicates the product is a C++ compiler. This macro is valid for C++ programs only.
__DBCS__	Indicates DBCS support is enabled. Defined using the /Sn compiler option.
__DDNAMES__	Indicates ddnames are supported. Defined using the /Sh compiler option.
__DLL__	Indicates code for a DLL is being compiled. Defined using the /Ge- compiler option.

Predefined Macros

__FUNCTION__	Indicates the name of the function currently being compiled. For C++ programs, expands to the actual function prototype.
__IBMC__	Indicates the version number of the C/C++ Tools C compiler.
__IBMCPP__	Indicates the version number of the C/C++ Tools C++ compiler.
_M_I386	Indicates code is being compiled for a 386 chip or higher.
__MULTI__	Indicates multithread code is being generated. Defined using the /Gm compiler option.
__OS2__	Set to the integer 1. Indicates the product is an OS/2 compiler.
__SPC__	Indicates the subsystem libraries are being used. Defined using the /Rn compiler option.
__TEMPINC__	Indicates the template-implementation file method of resolving template functions is being used. Defined using the /Ft compiler option.
__TILED__	Indicates tiled memory is being used. Defined using the /Gt compiler option.
__32BIT__	Set to the integer 1. Indicates the product is a 32-bit compiler.

The value of the __IBMC__ and __IBMCPP__ macros is 200, and is always defined. The macros __OS2__, _M_I386, and __32BIT__ are always defined also. The remaining macros, with the exception of __FUNCTION__, are defined when the corresponding #pragma directive or compiler option is used.

Intrinsic Functions

When optimization is on, the C/C++ Tools compiler by default generates code instead of a function call for the following C library functions:

abs	labs	memmove	strchr	strncat
_clear87	memchr	memset	strcmp	strncmp
_control87	memcmp	_status87	strcpy	strncpy
fabs	memcpy	strcat	strlen	strrchr

When you #include the appropriate header file in which the function prototype and the #define and #pragma statements for the function are found, the C/C++ Tools compiler generates code instead of a function call for these functions.

You can override the header either by undefining the macro or by placing the name of the function in parentheses, thus disabling the processor substitution. The function then remains a function call, and is not replaced by the code. The size of your object module is reduced, but your application program runs more slowly.

Note: The following functions are built-in functions, meaning they do not have any backing library functions, and are **always** inlined:

_alloca	_fasin	_fsin	_f2xm1	_interrupt
_disable	_fcos	_fsincos	_getTIBvalue	_outp
_enable	_fcossin	_fsqrt	_inp	_outpd
_facos	_fpatan	_fyl2x	_inpd	_outpw
	_fptan	_fyl2xp1	_inpw	__parmdwords

The built-in functions are all defined in <**builtin.h**>, in addition to the standard header definitions.

Intrinsic Functions

Appendix C. Mapping

This appendix describes how the C/C++ Tools compiler maps data types into storage and the alignment of each data type and the mapping of its bits. The mapping of identifier names is also discussed, as is the encoding scheme used by the compiler for encoding or *mangling* C++ function names.

Name Mapping

To prevent conflicts between user-defined identifiers (variable names or functions) and C/C++ Tools library functions, do not use the name of any library function or external variable defined in the library as a user-defined function.

If you statically link to the C/C++ Tools runtime libraries (using the /Gd- option), all external names beginning with Dos, Vio, or Kbd (in the case given) become reserved external identifiers. These names are not reserved if you dynamically link to the libraries.

To prevent conflicts with internal names, do not use an underscore at the start of any of your external names; these identifiers are reserved for use by the compiler and libraries. The internal C/C++ Tools identifier names that are not listed in either the *C Language Reference* or this manual all begin with an underscore (_).

If you have an application that uses a restricted name as an identifier, change your code or use a macro to globally redefine the name and avoid conflicts. You can also use the #pragma map directive to convert the name, but this directive is not portable outside of SAA.

A number of functions and variables that existed in the IBM C/2 and Microsoft C Version 6.0 compilers are implemented in the C/C++ Tools product, but with a preceding underscore to conform to ANSI naming requirements. When you run the C/C++ Tools compiler in extended mode (which is the default) and include the appropriate library header file, the original names are mapped to the new names for you. For example, the function name putenv is mapped to _putenv. When you compile in any other mode, this mapping does not take place.

Note: Because the name timezone is used as a structure field by the OS/2 operating system, the variable _timezone is **not** mapped to timezone.

Demangling (Decoding) C++ Function Names

When the C/C++ Tools compiler compiles a program, it encodes all function names and certain other identifiers to include type and scoping information. This encoding process is called *mangling*. The linker uses the mangled names to ensure type-safe linkage. These mangled names are used in the object files and in the final executable file. Tools that use these files must use the mangled names and not the original names used in the source code.

C/C++ Tools provides two methods of converting mangled names to the original source code names, demangling functions and the CPPFILT utility.

Using the Demangling Functions

The runtime library contains a small class hierarchy of functions that you can use to demangle names and examine the resulting parts of the name. It also provides a C-language interface you can use in C programs. The functions use no external C++ features.

The demangling functions are available in both the static (.LIB) and dynamic (.DLL) versions of the library. The interface is documented in the <**demangle.h**> header file.

Using the demangling functions, you can write programs to convert a mangled name to a demangled name and to determine characteristics of that name, such as its type qualifiers or scope. For example, given the mangled name of a function, the program returns the demangled name of the function and the names of its qualifiers. If the mangled name refers to a class member, you can determine if it is static, const, or volatile. You can also get the whole text of the mangled name.

Demangling C++ Function Names

To demangle a name, which is represented as a character array, create a dynamic instance of the Name class and provide the character string to the class's constructor. For example, to demangle the name f__1XFi, create:

```
char   rest;
 Name  name = Demangle("f__1XFi", rest);
```

The demangling functions classify names into five categories: function names, member function names, special names, class names, and member variable names. After you construct an instance of class Name, you can use the Kind member function of Name to determine what kind of Name the instance is. Based on the kind of name returned, you can ask for the text of the different parts of the name or of the entire name.

For the mangled name f__1XFi, you can determine:

```
name->Kind() == MemberFunction
((MemberFunctionName  ) name)->Scope()->Text() is "X"
((MemberFunctionName  ) name)->RootName() is "f"
((MemberFunctionName  ) name)->Text() is "X::f(int)"
```

If the character string passed to the Name constructor is not a mangled name, the Demangle function returns NULL.

For further details about the demangling functions and their C++ and C interfaces, refer to the information contained in the <**demangle.h**> header file. If you installed using the defaults, this header file should be in the INCLUDE directory under the main C/C++ Tools installation directory.

Using the CPPFILT Utility

The C/C++ Tools product also provides the CPPFILT utility to convert mangled names to demangled names. You can run this utility on your object file to produce a list of symbols that are contained in the file. The list includes both the mangled and demangled names.

One of the applications of this utility is creating module definition files for your C++ DLLs. Because functions in the DLL have mangled names, when you list the EXPORTS in your .DEF, you must use the mangled names. You can use the CPPFILT utility to extract all the names from the object module for you, copy the ones you want to export into your .DEF file, and link your object module into a DLL.

For more information on how to use the CPPFILT utility, see the READ.ME file in the main C/C++ Tools directory.

Data Mapping

The following section lists each data format and its equivalent C type in the C/C++ Tools product, including the alignment and mapping for each.

Automatic Variables: When optimization is turned off (/O-), automatic variables have the same mapping as other variables, but they are mapped on the stack instead of in a data segment. Because memory on the stack is constantly reallocated on the stack, **automatic variables are not guaranteed to be retained after the return of the function that used them**.

When optimization is on, automatic variables are mapped as follows:

Size of Object	Alignment
1-byte	Byte-aligned
2-byte	Word-aligned
3- to 4-byte	Doubleword-aligned
5- to 8-byte	8-byte-aligned
Greater than 8-byte	16-byte aligned.

Note that the variables are ordered to minimize padding.

In the C/C++ Tools product, a *word* consists of 2 bytes (or 16 bits) and a *doubleword* consists of 4 bytes (32 bits).

1. **Single-Byte Character**

Type	signed char and unsigned char
Alignment	Byte-aligned.
Storage mapping	Stored in 1 byte.

2. **Two-Byte Integer**

   Type: short and its signed and unsigned counterparts

   Alignment: Word-aligned.

   Storage mapping: Byte-reversed, for example, x3B2C (where 2C is the least significant byte and 3B is the most significant byte) is represented in storage as:

byte 0	**byte 1**
2C	3B

   Toward high memory →

3. **Four-Byte Integer**

   Type: long, int, and their signed and unsigned counterparts

   Alignment: Doubleword-aligned.

   Storage mapping: Byte-reversed, for example, x4A5D3B2C (where 2C is the least significant byte and 4A is the most significant byte) is represented in storage as:

byte 0	**byte 1**	**byte 2**	**byte 3**
2C	3B	5D	4A

   Toward high memory →

Note on IEEE Format

In IEEE format, a floating point number is represented in terms of sign (S), exponent (E), and fraction (F):

$(-1)^S \times 2^E \times 1.F$

In the diagrams that follow, the first two rows number the bits. Read them vertically from top to bottom. The last row indicates the storage of the parts of the number.

4. **Four-Byte Floating Point (IEEE Format)**

Type
: float

Alignment
: Doubleword-aligned.

Bit mapping
: In the internal representation, there is 1 bit for the sign (S), 8 bits for the exponent (E), and 23 bits for the fraction (F). The bits are mapped with the fraction in bit 0 to bit 22, the exponent in bit 23 to bit 30, and the sign in bit 31:

```
3 32222222 2221111111111
1  9876543 21 987654321 987654321

S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFF
```

Storage mapping
: The storage mapping is as follows:

byte 0	byte 1	byte 2	byte 3
76543210	111111 54321098	22221111 32109876	33222222 10987654
FFFFFFFF	FFFFFFFF	EFFFFFFF	SEEEEEEE

Toward high memory →

5. **Eight-Byte Floating Point (IEEE Format)**

Type
: double

Alignment
: Doubleword-aligned on the 80386

Bit mapping
: In the internal representation, there is 1 bit for the sign (S), 11 bits for the exponent (E), and 52 bits for the fraction (F). The bits are mapped with the fraction in bit 0 to bit 51, the exponent in bit 52 to bit 62, and the sign in bit 63:

```
6 66655555555 5544444444443333333333222222222211111111111
3 21 98765432 1 987654321 987654321 987654321 987654321 987654321

S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Storage mapping
: The storage mapping is as follows:

byte 0	byte 1	byte 2	...
76543210	111111 54321098	22221111 32109876	...
FFFFFFFF	FFFFFFFF	FFFFFFFF	...

Toward high memory →

byte 5	byte 6	byte 7
44444444 76543210	55555544 54321098	66665555 32109876
FFFFFFFF	EEEEFFFF	SEEEEEEE

Toward high memory →

6. **Ten-Byte Floating Point in Sixteen-Byte Field (IEEE Format)**

Type: long double

Alignment: Doubleword-aligned on the 80386

Bit mapping: In the internal representation, there is 1 bit for the sign (S), 15 bits for the exponent (E), and 64 bits for the fraction (F). The bits are mapped with the fraction in bit 0 to bit 63, the exponent in bit 64 to bit 78, and the sign in bit 79:

```
7 777777777666666
9 87654321 987654

S EEEEEEEEEEEEEEE

6666555555555544444444443333333333222222222211111111111
321 987654321 987654321 987654321 987654321 987654321 987654321

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

Storage mapping: The storage mapping is as follows:

byte 0	byte 1	byte 2	...
76543210	111111 54321098	22221111 32109876	...
FFFFFFFF	FFFFFFFF	FFFFFFFF	...

Toward high memory →

byte 7	byte 8	byte 9
66666555 43210987	77666666 10987654	77777777 98765432
FFFFFFFF	EEEEEEEE	SEEEEEEE

Toward high memory →

7. **Null-Terminated Character Strings**

Type
: char string[n]

Size
: Length of string (not including null).

Alignment
: Byte-aligned. If the length of the string is greater than a doubleword, the string is doubleword-aligned.

Storage mapping
: The string "STRING" is stored in adjacent bytes as:

byte 0	**byte 1**	**byte 2**	**byte 3**	**byte 4**	**byte 5**	**byte 6**
'S'	'T'	'R'	'I'	'N'	'G'	'\ '

Toward high memory →

8. **Fixed-Length Arrays Containing Simple Data Types**

Type
: The corresponding C/C++ Tools declaration depends on the simple data type in the array. For an array of int, for example, you would use something like:

```
int  int_array[n];
```

: For an array of float, you would use something like:

```
float  float_array[n];
```

Size
: $n\ \ (s + p)$, where *n* is the number of elements in the array, *s* is the size of each element, and *p* is the alignment padding.

Alignment
: The alignment is the same as that of the simple data type of the array elements. For instance, an array of short elements would be word-aligned, while an array of int elements would be doubleword-aligned. If the length of the array is greater than a doubleword, the array is doubleword-aligned.

Storage mapping
: The first element of the array is placed in the first storage position. For multidimensional arrays, row-major ordering is used.

9. **Aligned Structures**

   Type: struct

   Size: Sum of the sizes for each type in the struct plus padding for alignment.

   Alignment: The first element of the structure is aligned according to the alignment rule of the element that has the most restrictive alignment rule. If the length of the structure is greater than a doubleword, the structure is doubleword-aligned. The alignment of the individual members is not changed. In the following example, types char, short, and float are used in the struct. Because float must be aligned on the doubleword boundary, and because this is the most restrictive alignment rule, the first element must be aligned on the doubleword boundary even though it is only a char.

   Note: The first element will not necessarily occupy a doubleword, but it will be aligned on it.

```
struct y {
        char char1;        /  aligns on doubleword  /
        short short1;      /  aligns on word  /
        char char2;        /  aligns on byte  /
        float float1;      /  aligns on doubleword  /
        char char3         / aligns on byte  /
      };
```

Storage mapping The struct is stored as follows:

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5
char1	pad	short1	short1	char2	pad

Toward high memory →

byte 6	byte 7	byte 8	byte 9	byte 10
pad	pad	float1	float1	float1

Toward high memory →

byte 11	byte 12	byte 13	byte 14	byte 15
float1	char3	pad	pad	pad

Toward high memory →

Note: This mapping is also true for aligned structures in C++ as long as the structure does not contain virtual base classes or virtual functions.

10. **Unaligned or Packed Structures**

Type The definition of the structure variable is preceded by the keyword _Packed, or the #pragma pack directive or /Sp option is used. For instance, the following definition would create a packed struct called mystruct with the type struct y (defined above):

```
_Packed struct y mystruct
```

Size The sum of the sizes of each type that makes up the struct.

Data Mapping

Storage mapping

When the _Packed keyword, the #pragma pack(1) directive, or /Sp(1) option is used, the structure mystruct is stored as follows:

byte 0	byte 1	byte 2	byte 3	byte 4
char1	short1	short1	char2	float1

Toward high memory →

byte 5	byte 6	byte 7	byte 8
float1	float1	float1	char3

Toward high memory →

When #pragma pack(2) or the /Sp(2) option is used, mystruct is stored as follows:

byte 0	byte 1	byte 2	byte 3	byte 4	byte 5
char1	pad	short1	short1	char2	pad

Toward high memory →

byte 6	byte 7	byte 8	byte 9	byte 10	byte 11
float1	float1	float1	float1	char3	pad

Toward high memory →

Note: This mapping is also true for aligned structures in C++ as long as the structure does not contain virtual base classes or virtual functions.

11. **Arrays of Structures**

Type	The definition for an array of struct would look like: struct y mystruct_array[n] The definition of an array of _Packed struct would look like: _Packed struct y mystruct_array[n]
Alignment	Each structure is aligned according to the structure alignment rules. This may cause a fixed-length gap between consecutive structures. In the case of packed structures, there is no padding.
Storage mapping	The first element of the array is placed in the first storage position. Row-major ordering is used for multidimensional arrays. **Note:** This mapping is also true for aligned structures in C++ as long as the structure does not contain virtual base classes or virtual functions.

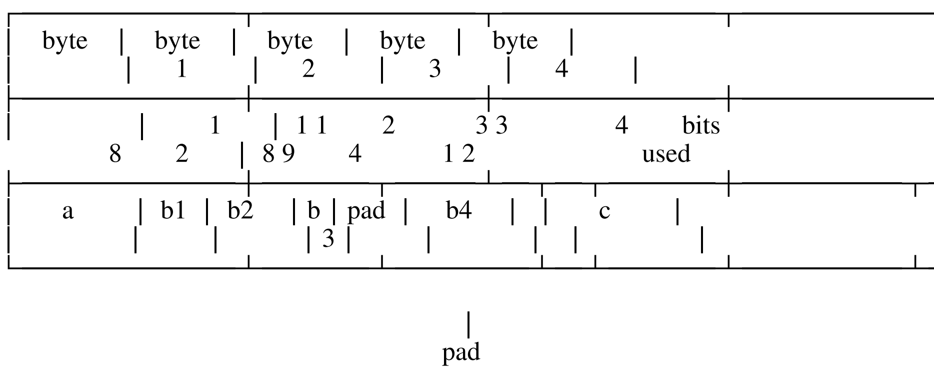
12. **Structures Containing Bit Fields**

Type	struct
Size	The sum of the sizes for each type in the struct plus padding for alignment.
Alignment	Each structure is aligned according to the structure alignment rules.

Storage mapping Given the following structure:

```
struct s {
         char  a;
         int  b1:4;
         int  b2:6;
         int  b3:1;
         int     : ;
         int  b4:7;
         char  c;
        }
```

`struct s` would be stored as follows:



Notes:

a. The second row of the table counts the number of bits used and should be read vertically top-to-bottom.

b. This mapping is also true for aligned structures in C++ as long as the structure does not contain virtual base classes or virtual functions.

Appendix D. Solving Common C Problems

This appendix contains possible solutions to common and often-reported C and C/C++ Tools problems. If you have a problem with your C program, look here first for a solution. If these questions included here do not describe your problem, or if the answers do not solve it, please refer to “If You Need More Help” on page 429 for information on how to get more help.

The questions have been grouped into the following sections:

The questions in each section are printed in bold type, followed immediately by a possible solution.

Writing a Program

Q:
When do you use the & and operators on pointers and arrays?

A:
The address operator (&) is used to return a pointer to the location of the operand; the indirection operator ( ) is used to access the data object that is pointed to. For example, if pSample is defined as a pointer to type int and a is defined as an int:

```
int  pSample;
int  a;
```

the following statements together assign the value 3 to a:

```
pSample   = &a;        /  pSample now points to variable a                  /
 pSample = 3;          /  whatever pSample points to gets the value 3  /
```


When you pass an array variable to a function, remember that the name of an array evaluates to a pointer to the first element in the array. When you declare the function, do not specify the operator; when you call the function, do not use the & operator. For example, the following statements do not process the array correctly:

```
int intarray[1 ][2];                    /  Array[1 ] of Array[2] of int  /
int foo(int ( argarray) [1 ][2] );

result = foo( &intarray );
```

To pass the array correctly, code the function definition and the function call as follows:

```
int intarray[1 ][2];                    /  Array[1 ] of Array[2] of int  /
int foo(int argarray[1 ][2]);       /  No   in this statement  /

result = foo( intarray );              /  No & in this statement  /
```

For more information on pointers, arrays, and the & and operators, see the *Online Language Reference*.

Q:
How do you use the ## operator?

A:
The ## operator is used in a macro invocation to concatenate two tokens to form a single token.

When you use ##, concatenation takes place before any individual arguments are expanded. Also, if the result of a concatenation contains valid macro names, further macro expansions can take place.

For example, consider the following code fragment:

```
#define ab a
#define p(x,y) x ## y
#define p1(x,y) p(x,y)

p1(ab,b);
```

The macro expansion for this code occurs in the following sequence:

1. Macro p1(x,y) is invoked, with parameter x associated with argument ab and parameter y associated with argument b.
2. Macro ab is invoked, replacing ab with a.
3. Macro p(x,y) is invoked, with parameter x associated with argument a and parameter y associated with argument b. This concatenates the two arguments and produces ab as a result.
4. Macro ab is invoked, replacing ab with a Because this is the last expansion that can be performed, a is returned as the result of the entire macro expansion.

For more information and examples on the use of the ## operator, see the *Online Language Reference*.

Compiling a Program

Q:
The compiler cannot find the os2.h **file.**

A:
The os2.h file is part of the Toolkit.

Make sure that you have installed the Toolkit, and that you specify the TOOLKT2 \INCLUDE directory in the INCLUDE environment variable.

Q:
Why does the compiler generate an error message for the statement
((long)fred)++**?**

A:
The operand of the increment operator (++) must be an lvalue. Because a cast does not produce an lvalue, the statement above does not compile.

Operators that must have lvalue operands include the increment and decrement operators ++ and --, as well as the simple and compound assignment operators.

Use the following statement instead:

```
(&((long  )fred))++
```

For more information on casting and lvalues, see the *Online Language Reference*.

Q:
Why does the compiler generate an incomplete type error message or a type mismatch error message for the following declarations?

```
 int f(struct st);
 struct st {int s1;} ss;

f(ss);
```

A:
A structure declaration must appear before any function prototype statements that use that structure type. In the above example, because struct st is declared after the function prototype for f, the compiler considers the declaration of struct st to be a new declaration. When function f is called, the compiler generates an error that the variable ss is not the same type as the parameter in the function prototype.

Change the order of the statements so that the structure declaration appears before the prototype. For example:

```
 struct st {int s1;} ss;
 int f(struct st);

f(ss);
```

Linking a Program

Q:
The linker generates unresolved external errors.

A:
The linker probably cannot find the libraries it needs to construct the executable module.

Make sure that you specify all necessary libraries when you invoke the linker. The correct C/C++ Tools libraries are linked in by default, unless you use the /Gn compiler option or the /NOD linker option.

Avoid using the /Gn compiler option and the /NOD linker option. The /Gn option suppresses information about the default libraries from the linker; the /NOD option causes the linker to ignore the default libraries. When you use these options you must specify on the command line all libraries you use, both directly and indirectly.

Q:
The linker cannot find the OS2386.LIB file.

A:
The OS2386.LIB file is part of the Toolkit.

Make sure that you have installed the Toolkit, and that you specify the TOOLKT20\LIB directory in the LIB environment variable.

Q:
My program uses a function uppercase, which is defined in a second source file as follows:

```
#include <ctype.h>
 void UPPERCASE (char   lower)
{
     while  ( lower)
     {
          lower = toupper ( lower);
        lower++;
     }
}
```

When I use icc to compile and link the two files, I get a linker error that says uppercase is an unresolved external. If I compile them and then invoke the linker in a separate step, everything works as it should.

A:
Your function name is defined in uppercase, and you call it in lowercase. Because the C language is case-sensitive, icc passes the /NOI option to the linker to make it case-sensitive also. You cannot disable this option.

You can continue to compile and link in separate steps, but because the C/C++ Tools libraries and most other C code is case-sensitive, you may encounter more problems at a later time. It is recommended you change your code so that the case of the function name is the same in the definition and in the call.

Running a Program

This section contains possible answers to questions that you might have when you run a program. Refer to this section if your program ends abnormally or behaves unexpectedly. This section is divided into the following topics:

Problems with DLLs

Q:
My DLL does not work properly.

A:
| You may be mixing objects compiled with the /Ge+ and /Ge- compiler options in the same DLL.

| The C/C++ Tools libraries provide different initialization routines for executable modules and DLLs. To ensure that the correct initialization routine is run, use the /Ge+ option is used when you create an executable module and the /Ge- option when you create a DLL.

| When you link your files, you can override the /Ge option you specified at compiler time. See "Using the /Ge Option" on page 118 for more information on how to do this. See Chapter 12, "Building Dynamic Link Libraries" on page 195 for more information on DLLs in general.

Q:
My DLL ends abnormally when a second process tries to call it.

A:
Make sure that you include the following statements in your module definition (.DEF) file:

LIBRARY INITINSTANCE TERMINSTANCE

> This statement identifies the executable file as a DLL. The INITINSTANCE attribute specifies that the _DLL_InitTerm function is called the first time the DLL is loaded for each process that accesses the DLL. The TERMINSTANCE attribute specifies that the _DLL_InitTerm function is called the last time the DLL is loaded for each process that accesses the DLL.

DATA MULTIPLE NONSHARED

> This statement defines the default attributes for data segments within the DLL. The MULTIPLE attribute specifies that there is a separate copy of the data segment for each process that accesses the DLL. The NONSHARED attribute specifies that the data segment is not shared by other processes.

For more information on DLLs, see Chapter 12, "Building Dynamic Link Libraries" on page 195.

Problems with Files

Q:
I can edit a file, but I cannot open it.

A:
Make sure that if you store the file name in a constant string, you use a double backslash (\\) to represent a backslash (\).

In C, a backslash is an escape character for inserting a character that you normally cannot type. For example, because \t is the tab character, the following string:

```
char filename[] = "c:\directory\test.c"
```

has an actual value of

```
"c:directory        est.c"
```

To enter the file name correctly, convert the string to the following:

```
char filename[] = "c:\\directory\\test.c"
```

Problems with Functions

Q:
I assigned a value to a function's parameter, but the value is not returned to the calling function.

A:
In C, parameters to a function are passed by value, not by reference. You need to pass either a pointer to the value (using the   operator) or the value's address (using the & operator).

For example, the following function x adds 2 integers and assigns the results to one of them:

```
void x(int a, int b)
{
    a = a + b;
}
```

Consider the following program:

```
#include  <stdio.h>

int  main(void)
{
      int c = 1 ;
      x( c, 5 );
      printf("The value of c is %d. \n", c);
}
```

Because c is passed by value, it is not changed to 15 as expected. It retains the value 1 after x returns.

To make the function work correctly, the parameter a must be defined with the operator:

```
 void x(int  a, int b)
{
       a =  a + b;
}
```

and the call must be made using the & operator:

```
 x( &c, 5 );
```

Q:
My function is not being called.

A:
Make sure that you include the parentheses () after the function name.

The parameter list enclosed in parentheses indicates that a function is to be called. This parameter list can be empty, but you must include the parentheses to actually invoke the function. If you use a function name by itself, without the parentheses, the statement only computes the address of the function.

Q:
Some of the old C code that I have recently started to use does not seem to work properly. It seems that the parameters to a function are not being received correctly.

A:
Make sure that you do not mix functions defined under the old C standard (K&R) with functions defined under the ANSI standard. You can still define functions according to the K&R standard, but you cannot mix prototyped and unprototyped function definitions because of the difference in conversions.

Note: The C++ language requires that all functions have ANSI-style prototypes.

There are important differences between the standards in the way that functions are defined and processed by the compiler. Under the old standard, you define a function as follows:

```
int my_function( variable1, variable2, variable3 )
    int     variable1;
    float  variable2;
    short  variable3;
{
...
}
```

To make passing of parameters easier, variables of type char or short are converted to type int and variables of type float are converted to double.

The ANSI standard formally defines the function using a function prototype. With the prototype definition, you explicitly state the number and types of parameters each function receives. The corresponding ANSI definition of the function above is:

```
/ function prototype /
int my_function( int variable1, float variable2, short variable3 );

/ function declaration /
int my_function( int variable1, float variable2, short variable3 )
{
...
}
```

The ANSI standard also allows functions with a variable number of parameters, specified by following the fixed parameter list with an ellipse ("..."). Under this standard, fixed parameters of type char are converted to int, optional parameters of type char and short are converted to int, and optional parameters of type float are converted to double.

It is best to convert the function definitions to the ANSI standard. Prototyping your functions as described in the ANSI standard makes your code more portable. Defining a full prototype also gives the compiler and optimizer complete information about the types and sizes of the parameters. As a result, the compiler does not have to perform conversions to widened types or generate eyecatcher instructions for the function.

Q:
Information that I generate by calling one function is being altered after I call a second function.

A:
You may be returning the address of a local variable. If you call a function from within your program, do not rely on any of its local data after it returns. For example, given the following function:

```
 void a( void )
{
    int   x;

     x = b();                      /  x points to a variable local to b()  /
    ...
    c();
    ...
     printf( "%d", &x ); /  try to access x after c() has been called  /
}
```

The int variable that x points to may not exist after function c is called, causing an error on the printf statement.

Local data is stored temporarily on the stack, which may be used by the operating system. If the operating system or another function needs some of the stack space, it is likely that the original data will be overwritten. The cause of this problem can be difficult to isolate, because the demand for stack space is random and unpredictable.

To avoid this problem, declare the variables in the calling function or as global variables.

Q:
My window procedures end abnormally.

A:
Make sure that you prototype your window procedures to use the _System calling convention. You can do this by including the appropriate system header file from the Toolkit. You should also ensure your window procedures include the EXPENTRY keyword, as described in the Toolkit documentation.

The C/C++ Tools compiler uses the _Optlink calling convention by default, which is not compatible with the _System calling convention used by the OS/2 system to call window procedures. OS/2 APIs use _System linkage; _Optlink is used for C/C++ Tools library functions.

It is easiest to use the _System keyword (or, for C only, the #pragma linkage directive) to give individual functions _System linkage.

For more information on the calling conventions, see Chapter 14, "Calling Conventions" on page 237.

Problems with Library Functions

Q:
A call to a printf statement causes the wrong thing to be printed or my program to end abnormally.

A:
Make sure that the parameters you list in your format string match the parameters you are actually passing to the function.

Possible problems include:

- Passing a parameter of a different type than you have declared.

  For example, the printf function in the following code fragment expects a string variable, but is passed a variable of type int:

  ```
  int  a;
   printf( "%s", a );
  ```

  The correct printf call should read:

  ```
  printf( "%d", a );
  ```

Passing a parameter of a different size than you have declared.

For example, the format string in the following code fragment indicates that three variables of type int are expected, but the variables passed are of type long, int, and short:

```
long   l;
int    i;
short  s;
 printf( "%d %d %d", l, i, s );
```

Because it reads in the bytes from storage, this call could have unexpected results. The correct printf call should read:

```
 printf( "%ld %d %hd", l, i, s );
```

Note: The C/C++ Tools compiler allows you to mix an %ld conversion operator with an int variable, and a %d conversion operator with a long variable. For portability, ensure that your conversion operators and variable types match.

Passing a parameter by reference instead of by value.

For example, in the following code fragment, the printf function expects a variable of type int, but is passed the address of an int variable:

```
int  a;
 printf( "%d", &a );
```

The correct printf call should read:

```
 printf( "%d", a );
```

Q:
The scanf function does not behave as expected. Sometimes it does not wait for input, does not convert all input, or goes into an infinite loop.

A:
The scanf function works on streams of characters, not lines of input. It reads the characters from the specified input stream and formats them according to the conversion rules that you specify.

Here are some guidelines to follow when reading character input:

- Use scanf for machine-generated input only.
- Use a combination of the fgets and sscanf functions for user input.

  Note: Do not substitute the gets function for fgets. If you use gets, it is possible to overwrite the character array used to store the input, and cause memory problems. With fgets, you control the number of characters that the user can input.
- Check the return count from the scanf functions to see how many fields were processed.
- Read the descriptions of the various formats carefully. Some formats skip leading white space (for example, %d and %f) and others do not (for example, %c). Remember to include the new-line character.
- Remember that the scanf conversion characters are different from the printf conversion characters.

The following examples show how scanf works and illustrate some possible problems. All of the examples assume that the input is coming from the user.

The following statement reads an integer from the user:

```
scanf( "%d", &myint )
```

The program waits for you to enter a string of characters. If you enter:

```
25\n
```

the function reads the digits 2 and 5 and stops when it reads the first non-decimal digit, the new-line character (\n).

If you instead enter:

```
13  74\n
```

the function reads the digits 1 and 3 and stops when it reads the blank, which is the first non-decimal digit. The %d conversion skips any leading whitespace characters such as a blank, the tab character (\t), and the new-line character.

The 74\n remains in the input stream. Because the input stream is not empty, the next call to scanf will read directly from the stream and will not wait for user input.

It is possible to enter an infinite loop with a combination of scanf and unexpected user input. Here is an example:

The following code fragment reads a set of numbers until a negative number is entered.

```
answer = ;
i = ;
while (answer >= ) {
     scanf( "%d" , &answer );
     myarray[i] = answer;
    i++;
}
```

If you enter:

```
123XYZ\n
```

the first call to scanf reads 123 as a valid integer and stops at the X, leaving XYZ\n in the input stream. Because the input stream is not empty, the next call to scanf tries to read an integer from the stream. Because X is not an integer, scanf never progresses through the input stream, and you do not have the opportunity to enter new data. The result is an infinite loop.

For more information on the scanf function, see the *C Library Reference*.

Q:
When I call a library function, it does not work or it causes my program to end abnormally.

A:
Make sure that you use the #include preprocessor directive to include the library header file that contains the prototype statement for the library function. If you use the /Ms compiler option, you must include header files for all library functions you use.

Note: You can use the /Wpro compiler option to warn you about unprototyped library functions.

Q:
My program links correctly and no error messages are generated, but the calls to library functions and system APIs do not work like they should.

A:
Make sure that you are using the correct calling convention. Library functions must be called using _Optlink and OS/2 APIs must be called using _System. Include the appropriate header files to ensure that the functions and APIs you use are prototyped correctly.

Note: You can use the /Wpro compiler option to warn you about unprototyped library functions.

Problems with Macros

Q:
A statement in my C program behaves strangely. It deals with a combination of a macro and increment operators.

A:
When you use a macro, make sure that you know how it will be expanded. If you define a macro that repeats the input argument, problems might occur in combination with increment (++) and decrement (--) operators.

As an example, given the following macro toupp:

```
#define toupp(c) islower(c) ? _toupper(c) : (c)
```

The following statement is intended to copy every character of source into dest:

```
while ( dest++ = toupp( source++));
```

After the toupp macro is expanded, the actual statement that is executed is:

```
while ( dest++ = islower( source++) ? _toupper( source++) : ( source++));
```

This increments source twice each time the loop is done, which is not what was intended.

Q:
I have defined a macro, but it does not always produce the correct answer.

A:
Make sure that you use parentheses when you define the macro. You may get unexpected results if you use the macro in the same statement as other operators. The precedence rules of the other operators may interfere with the macro definition.

For example, given the following code:

```
#define DOUBLE(x) x+x

y = DOUBLE(2)+1;          /  assigns 5 to y  /
z = DOUBLE(2) 3;          /  assigns 8 to z  /
```

The last statement evaluates to 8 rather than 12 because it expands to z = 2 + 2   3. To prevent this problem, use parentheses when you define a macro. For example, the above macro would give the expected results if it were defined as:

```
#define DOUBLE(x) ((x) + (x))
```

Problems with Threads

Q:
In my program, threads other than thread one do not work correctly.

A:
Ensure that:

You use the /Gs- compiler option to generate stack probes (which is the default).

You use the /Gm compiler option to link with the multithread libraries.

If you started a thread with the DosCreateThread API, you call _endthread to end the thread and perform the necessary termination actions. If you used _beginthread to start the thread, _endthread is called implicitly when the thread ends.

Problems with One Statement

Q:

My program sometimes takes the wrong branch of an if statement. It should be processing the else clause, but it processes the then clause instead.

A:

Ensure that your test statement uses the equality operator (==), not the assignment operator (=).

Because the result of an assignment is the value assigned, the then clause is executed whenever the right-hand expression is not zero. The else clause is executed only when the right-hand expression is zero.

One way to check for this situation is to place the constant or expression on the left hand side of the == operator and the variable to be tested on the right-hand side. If the assignment operator is used, the compiler generates an error message.

Note: You can use the /Wcnd compiler option to warn you about possible problems in conditional expressions.

Q:

I have an assignment statement, but it does not seem to do anything. The variable retains its original value.

A:

Make sure that you use the assignment operator (=), not the equality operator (==).

You can use the == operator in the same place as the = operator, as in the following C statement:

```
i == 2;
```

However, this statement instructs the compiler to test if the value of i is equal to 2. Because nothing is done with the results, this statement has no effect on any variables.

Note: You cannot access this type of statement from the C/C++ Tools debugger because the C/C++ Tools optimizer discards any statements that have no effect.

Q:
I have one statement that does not seem to do anything.

A:
Make sure that you are not missing an end to a comment (/). In the following example, an ending comment is omitted, causing a statement to be skipped during the compilation:

```
/  This comment has an incorrect terminator  \
...
 here = (is > some) ? important : code;
...
/  This comment "accidentally" terminates the previous comment  /
 code = begins + working / fine->again;
```

Q:
Why does (i & x F == 5) **behave as** (i & (x F == 5)) **instead of** ((i & x F) == 5) **as I expected?**

A:
The equality operator (==) has a higher precedence than the bitwise AND operator (&).

When you construct statements with multiple operators, make sure you understand the precedence and associativity rules for each of the operators. Use parentheses to clarify the purpose of your code and make it easier to understand.

For more information on the rules for operator precedence and associativity, see the *Online Language Reference*.

Problems with Groups of Statements

Q:
I have a group of statements that do not seem to do anything.

A:
Make sure that you are not missing an end to a comment (/). For an explanation, see the preceding section.

Q:
A section of my code is not producing the expected results. The statements use multiple C operators.

A:
When you construct statements with multiple operators, make sure you understand the precedence and associativity rules for each of the operators.

The precedence rules for C operators are complex and may not be in the order that you would expect. For example, given the following statement:

```
if (x & y ==  )
```

because the == operator has a higher precedence than the & operator, the compiler interprets the statement as:

```
if (x & (y ==  ))
```

To avoid confusion, use parentheses to clarify the purpose of your code and to make it easier to understand.

For more information on the rules for operator precedence and associativity, see the *Online Language Reference*.

Q:
A section of my code is not producing the expected results. The statements use the ++ and -- operators.

A:
Make sure that your statements do not depend on side effects of the ++ and -- operators. For example, because the result of the following statement depends on when the ++ operator is evaluated and when the assignment is done, it may produce inconsistent results:

```
s[i++] = t[i];
```

The order of these operations depends on the compiler being used and possibly on the optimization requirements. One compiler may compute the source address first (the right-hand side of the statement), while another may compute the target address first (the left-hand side).

To produce consistent results, if you use the ++ or -- operators on a variable within an expression, make sure that the variable appears only once within the expression.

If You Don't Know Where to Start

Q:
My code looks correct and there are no compiler errors, but the program is not producing the expected results.

A:
Make sure that there is not a semicolon at the end of a for, do, or while statement.

When there is only one statement in the body of a loop, it is common to code the loop in the following style:

```
for (... ; ... ; ...)
    statement;
```

It is also a common error to accidentally add a semicolon to the end of the first line. For example:

```
for (i =  ; i < SOMENUMBER; i++);
    d[i] =  ;
```

The semicolon at the end of the for statement ends the body of the loop. The second line, which is the intended body of the loop, is executed only once.

To avoid this problem, make the body of the loop into a compound statement by enclosing it in braces ({}).

Q:
My program does not run and the operating system generates a SYS2070 error.

A:
The program could not access an external reference. The linker may not have been able to resolve all of the external references in your program.

You should always use the /NOI linker option to preserve the case of external names when you link your program. If you use icc to invoke the linker, this option is passed for you by default.

Q:
My PM application disappears without generating any messages.

A:
An exception has been generated and intercepted by an exception handler that has terminated the program. A machine-state dump is sent to stderr, but because the PM interface direct stderr to a null output device, you cannot see the error messages.

Use the _set_crt_msg_handle function to redirect stderr to a file. You will then be able to see the runtime messages, including exception messages and machine-state information. Alternatively, you can write your own exception handler to intercept the exception and handle it however you want.

Q:
My program does not work properly. Sometimes adding or removing statements changes how the program terminates or may even solve the problem temporarily. Using a debugger changes the symptoms.

A:
There are several possible solutions to your problem:

Make sure that you are calling functions correctly, that is, that you are not missing a parameter in a function call or passing a parameter with an incorrect type.

If a parameter is missing, the program may replace the parameter with arbitrary data in order to complete the function call. The C/C++ Tools compiler checks for missing parameters if you define your functions using function prototypes.

Note: You can use the /Wpro compiler option to warn you about missing prototypes.

If the incorrect type of parameter is used, the function misreads the parameter list.

Ensure that strings are terminated by a null byte.

When you initialize a string, you must include space for the null byte. For example,

```
char str1[3] = "ab"     /  allocates 'a', 'b', '\ '  /
char str2[3] = "abc"    /  allocates 'a', 'b', 'c'; no '\ '  /
```

When you use a string, do not overwrite the null byte.

Because the null byte is used to indicate where the string terminates, a string without the null byte can cause memory problems. If you use a function such as strcpy on a string without the terminating character, portions of the following memory may be overwritten, causing problems with the current program or programs that are using that memory space. Problems could appear immediately or only after the program is run several times.

Check your function return types.

The compiler assumes that a function declared without a return type returns an int. This could cause problems if your program is expecting a different return type. Prototype your functions to avoid this problem.

If you declare an array in one file and reference it in another file using extern, make sure that the extern statement has the same form as the declaration statement.

For example, the following declarations are not equivalent:

```
 / File 1: Global Data definitions  /
char  x[1  ];

 / File 2: Using the global data  /
 extern char  x;
```

In the second file, the compiler generates code that assumes that the address at x contains the address of the actual array. The correct definition in File 2 is:

```
/ File 2: Using the global data  /
extern char x[];
```

Ensure that you are not referencing beyond the last element of an array.

The first entry of an array is found at index   (for example, array[ ]). If you declare an array of size *n*, the array starts at element   and ends at element *n*-1.

The following code fragment references beyond the last array element:

```
char  stuff[1 ];
int    i;
...
 for (i =  ; i <= 1 ; i++) {        / test should have been i < 1   /
       stuff[i] = ' ';
}
```

Referencing beyond the last element in the array may overwrite memory locations and cause problems with variable data, functions, or the entire program.

Ensure you use the malloc and free library functions correctly.

The malloc function returns a pointer to an area of memory that is at least as large as you request. The free function releases memory previously allocated by malloc.

Make sure that only pointers returned by the malloc function are passed to the free function. To keep track of what memory is available, malloc stores information in a section of memory adjacent to the pointer that it returns. The free function uses this information to return the allocated space to the list of available memory.

The free function does not check the pointer that it receives. If free receives a pointer that was not set by malloc, memory problems can occur. For example, other programs may get unauthorized access to your data areas, program code, or parts of the operating system.

Also make sure that you do not use memory outside of the memory allocated by malloc.

To help you find possible problems with these functions, use the debug memory management functions, described in the *C Library Reference*.

If You Need More Help

If the information in this appendix does not apply to your problem, or you would like to report a product defect, you can contact IBM by several means:

- In North America, call 1-8 -547-1283 to obtain the local number for the OS/2 support Bulletin Board System (OS2BBS) in your area. Note that the bulletin board does not provide defect support.
- If you are a CompuServe user, you can access the OS2DF1 forum and go to Section 4 for C and Debugger information, or to Section 5 for C++ information.
- If you have an Internet ID, you can contact IBM at cset2@vnet.ibm.com.
- To report a product defect **only**, call 1-8 -237-5511 and identify yourself as a C/C++ Tools product user. A problem management report (PMR) will be created to reflect the problem, and you will be given a PMR number that you can use to track your problem.

Appendix E. Component Files

This appendix lists the component files of the C/C++ Tools product and indicates where they are installed on your hard drive, assuming you used the IBM-supplied defaults for the installation.

The directory structure created by the default C/C++ Tools installation are as follows:

```
───IBMCPP (main directory)
    ├───READ.ME
    ├───BIN
    ├───DLL
    ├───HELP
    ├───IBMCLASS
    ├───INCLUDE
    │   └───SYS
    ├───LIB
    ├───LOCALE
    ├───SAMPLES
    │   ├───ICLCC
    │   ├───ICLUI
    │   ├───SAMPLE1A
    │   ├───SAMPLE1B
    │   ├───SAMPLE 2
    │   ├───SAMPLE 3
    │   ├───SAMPLE 4
    │   ├───SAMPLE 5
    │   └───SAMPLE 6
    ├───SYS
    ├───TMP
    ├───TUTORIAL
    │   ├───BROWSER
    │   └───DEBUG
    └───WKFRAME
       ├───DDE4CSET.PRF
       ├───GREP
       ├───HELLO2
       ├───MAHJONGG
       ├───PMLINES
       └───TOUCH
```

If you install the C/C++ Tools product on a LAN, your local directory will contain only the files CSETENV.CMD and DDE4XTRA.SYS.

Note: For the most current information about the directory structure and files, refer to the READ.ME file.

C/C++ Tools Files

This section lists the C/C++ Tools files by directory.

Note: The naming conventions used for the libraries are intended to help identify their function. The library names are as follows:

Character Position 1234 5 6 7 8	Significance
DDE4	Product prefix
S M N (position 5)	Single-thread library Multithread library Subsystem library (no environment)
B (position 6)	Builds both executables and DLLs
S (position 7)	Standard library
I O (position 8)	Import library Object library (contains initialization routines)

The files are as follows:

1. BIN

   CSETENV.CMD

   This file contains the commands to set the environment variables for the C/C++ Tools product. If you use the installation defaults, CSETENV.CMD contains the following statements:

```
@REM  DEVICE=C:\IBMCPP\SYS\DDE4XTRA.SYS
@REM  LIBPATH=C:\IBMCPP\DLL;
SET  PATH=C:\IBMCPP\BIN;%PATH%
SET  DPATH=C:\IBMCPP\LOCALE;C:\IBMCPP\HELP;C:\IBMCPP\SYS;%DPATH%
SET  LIB=C:\IBMCPP\LIB;%LIB%
SET  INCLUDE=C:\IBMCPP\INCLUDE;C:\IBMCPP\IBMCLASS;%INCLUDE%
SET  HELP=C:\IBMCPP\HELP;%HELP%
SET  BOOKSHELF=C:\IBMCPP\HELP;%BOOKSHELF%
SET  HELPNDX=DDE4C.NDX+DDE4CPP.NDX+DDE4CCL.NDX+DDE4UIL.NDX+%HELPNDX%
SET  TMP=C:\IBMCPP\TMP
SET  TZ=EST5EDT, , , , , , , ,
```

   | Note that the LIBPATH variable and DEVICE statement **must** be set in your CONFIG.SYS file. They cannot be set using a command file such as CSETENV.CMD.

   The compiler itself (ICC.EXE, DDE4FE.EXE, DDE4CPP.EXE DDE4BE.EXE, DDE4BE0.EXE)

   | The intermediate code linker (DDE4ICL.EXE)

   | The code for resolving template names (DDE4MNCH.EXE)

   The debugger (IPMD.EXE)

   | The browser (IBRS.EXE)

   | The execution trace analyzer (IXTRA.EXE, IDCGRAPH.EXE,
   | ICALNEST.EXE, IEXCDENS.EXE, ISTATS.EXE, ITIME.EXE)

2. DLL

Dynamic link libraries for the compiler:

- DDE4ICC.DLL - Compiler options DLL (for use with the WorkFrame/2 product).
- DDE4ICL.DLL - Linker options DLL (for use with the WorkFrame/2 product).
- DDE4MBS.DLL - Multithread standard DLL.
- DDE4NBS.DLL - Subsystem (no environment) DLL.
- DDE4SBS.DLL - Single-thread standard DLL.
- DDE4MNGL.DLL - DLL for name mangling and demangling.

DLLs for the debugger:

- DDE4BE32.DLL
- DDE4CRT.DLL
- DDE4CRTP.DLL
- DDE4CXT.DLL
- DDE4CXTP.DLL
- DDE4MODL.DLL
- DDE4PMDB.DLL
- DDE4RESS.DLL

DLLs for EXTRA:

- DDE4NARC.DLL
- DDE4XAPI.DLL
- DDE4XELV.DLL
- DDE4XTRA.DLL
- _DOSCALL.DLL
- FCLDLGP.DLL
- FCLDRCP.DLL
- _PMGPI.DLL
- _PMWIN.DLL

DLLs for the browser:

- XELV.DLL
- XARC.DLL

DLLs for the User Interface class library:

- ICRES437.DLL

3. HELP

Message files:

- DDE4.MSG - Runtime messages
- DDE4BE32.MSG - Debugger messages
- DDE41.MSG - Compiler back end messages
- DDE42.MSG - Compiler icc messages
- DDE43.MSG - Compiler front end messages
- DDE44.MSG - Intermediate code linker messages
- DDE45.MSG - C++ compiler front end messages
- DDE46.MSG - C++ Standard class library messages

Online help files:

- DDE4ICC.HLP - Help for compiler options (for use with the WorkFrame/2 product).
- DDE4ICL.HLP - Help for linker options (for use with the WorkFrame/2 product).
- DDE4HELP.HLP - Help for the debugger.
- DDE4XTRA.HLP - Help for EXTRA.
- DDE4BRS.HLP - Help for the browser.

Online references:

- DDE4SCL.INF - *Standard Class Library Reference* for the Complex Mathematics, I/O Stream, and Task libraries.
- DDE4CCL.INF - *Collection Class Library Reference* for the Collection class library.
- DDE4CLIB.INF - *C Library Reference* for all C/C++ Tools library functions.
- DDE4LRM.INF - *Online Language Reference* for C and C++ language constructs, compiler options, and messages.
- DDE4UIL.INF - *User Interface Class Library Reference* for the User Interface class library.

Files to enable context-sensitive help in the Enhanced editor (EPM):

- DDE4ERRS.HLP - Help for compiler messages (for use with the WorkFrame/2 product).
- DDE4C.NDX - Mapping file for C.
- DDE4CPP.NDX - Mapping file for C++
- DDE4CCL.NDX - Mapping file for the Collection class library.
- DDE4UIL.NDX - Mapping file for the User Interface class library.

4. INCLUDE

Runtime library header files:

<**assert.h**>	<**locale.h**>	<**stddef.h**>
<**builtin.h**>	<**malloc.h**>	<**stdio.h**>
<**conio.h**>	<**math.h**>	<**stdlib.h**>
<**ctype.h**>	<**new.h**>	<**string.h**>
<**demangle.h**>	<**memory.h**>	<**sys\stat.h**>
<**direct.h**>	<**process.h**>	<**sys\timeb.h**>
<**errno.h**>	<**search.h**>	<**sys\types.h**>
<**fcntrl.h**>	<**setjmp.h**>	<**sys\utime.h**>
<**float.h**>	<**share.h**>	<**terminat.h**>
<**io.h**>	<**signal.h**>	<**time.h**>
<**limits.h**>	<**stdarg.h**>	<**unexpect.h**>
		<**wcstr.h**>

C++ Standard Class Library header files:

<**complex.h**>	<**iomanip.h**>	<**stream.h**>
<**fstream.h**>	<**iostream.h**>	<**strstrea.h**>
<**generic.h**>	<**stdiostr.h**>	<**task.h**>

User Interface class library header files.

Collection class library header files.

5. LIB

Static runtime libraries for building both DLLs and executable modules:

- COMPLEX.LIB - Statically bound, single-thread Complex Mathematics Library.
- COMPLEXM.LIB - Statically bound, multithread C++ Complex Mathematics Library.
- DDE4MBS.LIB - Statically bound, multithread standard library.
- DDE4NBS.LIB - Statically bound, subsystem library.
- DDE4SBS.LIB - Statically bound, single-thread standard library.
- TASK.LIB - Statically bound, single-thread C++ Task Library.

Import libraries for building both DLLs and executable modules:

- DDE4MBSI.LIB - Dynamically bound, multithread standard import library.
- DDE4NBSI.LIB - Dynamically bound, subsystem import library.
- DDE4SBSI.LIB - Dynamically bound, single-thread standard import library.

Object libraries containing necessary startup routines:

- DDE4MBSO.LIB - Statically bound, multithread standard object library.
- DDE4NBSO.LIB - Statically bound, subsystem object library.
- DDE4SBSO.LIB - Statically bound, single-thread standard object library.

Static libraries for EXTRA:

- _DOSCALL.LIB
- _PMGPI.LIB
- _PMWIN.LIB

Static libraries for the User Interface class library:

- IBASE.LIB
- IBASEAPP.LIB
- IBASECTL.LIB
- ICNR.LIB
- IDRAG.LIB

Static libraries for the Collection class library:

- ICLCC.LIB

Object to link into your program to enable you to pass global file-name arguments to main (SETARGV.OBJ)

Object to link into your program for EXTRA (DDE4XTRA.OBJ)

6. IBMCLASS

   Header files for the Collection class library.
   Header files for the User Interface class library.

7. LOCALE

   Locale object files:

   - IBMCCDEF.CLD
   - IBMCFRAN.CLD
   - IBMCGERM.CLD
   - IBMCITAL.CLD
   - IBMCJAPN.CLD
   - IBMCJAP2.CLD
   - IBMCJAP3.CLD
   - IBMCSPAI.CLD
   - IBMCUK.CLD
   - IBMCUSA.CLD

8. SAMPLES

The ICLCC directory contains the sample programs for the Collection class library.

The ICLUI directory contains the sample programs for the User Interface class library.

The directories SAMPLE*nn* each contain the files for a sample program and two command files used to compile, link, and run the program. For example, the SAMPLE1A directory contains the files for the SAMPLE1A program.

9. SYS

The device driver for EXTRA (DDE4XTRA.SYS)
The profile for the browser (DDE4BRS.PRF)

10. TUTORIAL

The BROWSER directory contains the files for the browser tutorial.
The DEBUG directory contains the files used for the online debugger tutorial.
The other directories under TUTORIAL contain the files for the Collection class library tutorials.

11. TMP

This directory contains any temporary files created by the compiler.

12. WKFRAME

This directory contains the C/C++ Tools files provided for IBM WorkFrame/2 support.

- DDE4CSET.PRF - Language profile for the C/C++ Tools product.
- Each of the directories under WKFRAME contains the files for a sample project that can be created using the WorkFrame/2 and C/C++ Tools products. For example, the TOUCH directory contains the files for the TOUCH sample project.

Glossary

This glossary defines terms and abbreviations that are used in this book. It does not include all terms previously established in the *SAA CPI C Reference - Level 2*. If you do not find the term you are looking for, refer to the index or to the *IBM Dictionary of Computing*, SC20-1699.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

absolute value. The magnitude of a real number regardless of its algebraic sign.

| **abstract code unit (ACU)**. A measurement used by the C/C++ Tools compiler for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

| **access**. An attribute that determines whether or not a class member is accessible in an expression or declaration.

| **access declaration**. A declaration used to restore access to members of a base class.

| **access specifier**. One of the C++ keywords public, private, or protected.

| **ACU**. Abstract code unit.

| **address**. A name, label, or number identifying a location in storage, a device in a system or network, or any other data source.

aggregate. An array or a structure. Also, a compiler option to show the layout of a structure or union in the listing.

alias. An alternate label used to refer to the same data element or point in a computer program.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

Note: IBM has defined an extension to ASCII code (characters 128-255).

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

anonymous union. A union that is declared within a structure or class and that does not have a name.

ANSI. American National Standards Institute.

API. Application program interface.

application. The use to which an information processing system is put, for example, a payroll application, an airline reservation application, a network application.

application program interface (API). The formally defined programming language interface between an IBM system control program or a licensed program and the user of the program.

argument. In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called a parameter.

arithmetic object. An integral object, a bit field, or floating-point object.

array. A variable that contains an ordered group of data objects. All objects in an array have the same data type.

ASCII. American National Standard Code for Information Interchange.

assembly language. A symbolic programming language in which the set of instructions includes the instructions of the machine and whose data structures correspond directly to the storage and registers of the machine.

asynchronous. Without regular time relationship; unexpected or unpredictable with respect to the execution of program instructions.

B

base class. A class from which other classes are derived. A base class may itself be derived from another base class. See also *abstract class*.

binary. (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Involving a choice of two conditions, such as on-off or yes-no.

binary expression. An expression containing two operands and one operator.

binary stream. An ordered sequence of untranslated characters.

bit. A binary digit.

bit field. A member of a structure or union that contains a specified number of bits.

block. The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

block statement. Any number of data definitions, declarations, and statements that appear between the symbols { and }. The block statement is considered to be a single C-language statement.

boundary alignment. The position in main storage of a fixed-length field (such as byte or doubleword) on an integral boundary for that unit of information. For the C/C++ Tools example, a word boundary is a storage address evenly divisible by two.

buffer. A portion of storage used to hold input or output data temporarily.

built-in. A function which the compiler will automatically inline instead of the function call unless the programmer specifies not to.

byte. For IBM C compilers, 8 bits equal 1 byte.

C

C/2. Pertaining to a version of the C language designed for the OS/2 environment.

call. To transfer control to a procedure, program, routine, or subroutine.

catch block. A block associated with a try block that receives control when a C++ exception matching its argument is thrown.

case clause. In a switch statement, a case label followed by any number of statements.

case label. The word case followed by a constant expression and a colon.

cast. An expression that converts the type of the operand to a specified scalar data type (the operator).

character constant. A character or an escape sequence enclosed in single quotation marks.

character set. A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

child process. The new process created by a spawn or exec call.

| **class**. A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. Classes may be defined hierarchically, allowing one class to be an expansion of another, and may restrict access to its members.

| **class library**. A collection of C++ classes.

| **client program**. A program that uses a class. The program is said to be a client of the class.

| **Collection class library**. A set of classes that provide basic functions and can be used as base classes.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

comment. A comment contains text that the compiler ignores. Comments begin with the /* characters, end with the */ characters, and can span any number of lines. For C++ files, and for C files if the /Ss compiler option is used, // characters begin a comment which ends at the end of the line.

compile. To transform a set of programming language statements (source file) into machine instructions (object module).

compiler. A program that translates instructions written in a programming language (such as C language) into machine language.

| **Complex Mathematics library**. A class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

complex number. A number consisting of an ordered pair of real numbers, expressible in the form a+bi, where a and b are real numbers and i squared equals minus one.

condition. A relational expression in a program or procedure that can be evaluated to a value of either true or false.

const. An attribute of a data object that declares the object cannot be changed.

constructor. A special class member function that has the same name as the class and is used to construct and possibly initialize class objects.

control statement. A statement that changes the path of execution.

conversion. A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values. Because accuracy of data representation varies among different data types, information may be lost in a conversion.

D

data definition (DD). A statement that is stored in the environment and that externally identifies a file and the attributes with which it should be opened.

data definition name (ddname). The part of the data definition before the equal sign. It is the name used in a call to fopen or freopen to refer to the data definition stored in the environment.

data definition (DD) statement. Synonym for data definition.

data object. A storage area used to hold a value.

data stream. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format.

DBCS. (1) See *double-byte character set*. (2) See *ASCII*.

ddname. Data definition name.

decimal. A base ten number system; decimal digits range from 0 (zero) through 9 (nine).

declaration. A description that makes an external object or function available to a function or a block.

declare. To identify the variable symbols to be used at preassembly time.

default. An attribute, value or option that is used when no alternative is specified by the programmer.

default argument. An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default value is used. Arguments with default values must be the trailing arguments in a function prototype argument list.

default constructor. A constructor that takes no arguments, or for which all the arguments have default values.

define directive. A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. A data description that reserves storage and may provide an initial value.

definition (DEF) file. Synonym for module definition file.

delete. (1) A C++ keyword that identifies a free storage deallocation operator. (2) A C++ operator used to destroy objects created by new.

demangling. The conversion of mangled names back to their original source code names. See also *mangling*.

denormal. Pertaining to a number with a value so close to that its exponent cannot be represented normally. The exponent can be represented in a special way at the possible cost of a loss of significance.

destructor. A special member function that has the same name as its class, preceded by a tilde (˜), and that "cleans up" after an object of that class, for example, freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

digit. Any of the numerals from 0 through 9.

directory. A file containing the names and controlling information for other files or other directories.

DOS. Disk Operating System.

domain. All the possible input values for a function.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS capable.

double precision. Pertaining to the use of two computer words to represent a number with greater accuracy. For example, a floating-point number would be stored in the long format.

doubleword. A sequence of bits or characters that comprises two computer words and can be addressed as a unit. For the C/C++ Tools compiler, a doubleword is 32 bits (4 bytes).

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time.

E

EBCDIC. See *extended binary-coded decimal interchange code*.

E-format. Floating-point format, consisting of a number in scientific notation.

elaborated type specifier. A specifier typically used in an incomplete class declaration to qualify types that are otherwise hidden.

element. A data object in an array.

encapsulation. The hiding of the internal representation of data objects and implementation details from the client program.

enumeration constant. An identifier that is defined in an enumerator and that has an associated integer value. You can use an enumeration constant anywhere an integer constant is allowed.

enumeration data type. A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

EOF. End of file.

escape sequence. A representation of a character. An escape sequence contains the \ symbol followed by one of the characters: a, b, f, n, r, t, v, ', ", x, \, or followed by one to three octal or hexadecimal digits.

exception. (1) Under the OS/2 operating system, a user or system error detected by the system and passed to an OS/2 or user exception handler. (2) For C++, any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called throwing the exception).

exception handler. (1) Under the OS/2 operating system, a function that receives the OS/2 exception and either corrects the problem and returns execution to the program, or terminates the program. (2) In C++, a catch block that catches a C++ exception when it is thrown from a function in a try block.

exception handling. A type of error handling that allows control and information to be passed to an exception handler when an exception occurs. Under the OS/2 operating system, exceptions are generated by the system and handled by user code. In C++, try, catch, and throw expressions are the constructs used to implement C++ exception handling.

executable program. A program that can be run on a processor.

expression. A representation for a value. For example, variables and constants appearing alone or in combination with operators.

extended binary-coded decimal interchange code (EBCDIC). A set of 256 eight-bit characters.

extension. (1) An element or function not included in the standard language. (2) File name extension.

external data definition. A definition appearing outside a function. The defined object is accessible to all functions that follow the definition and are located within the same source file as the definition.

eyecatcher. A recognizable sequence of bytes that determine which parameters were passed in which registers. This sequence is used for functions that have not been prototyped, have a variable number of parameters, and use _Optlink linkage.

F

file. A collection of data that is stored and retrieved by an assigned name.

file handle. A value created by the system that identifies a drive, directory, and file so that the file can be found and opened.

file name. The name used to identify a file.

float constant. A constant representing a nonintegral number.

friend class. A class in which all the member functions are granted access to the private and protected members of another class. It is named in the declaration of the other class with the prefix friend.

friend function. A function that is granted access to the private and protected parts of a class. It is named in the declaration of the other class with the prefix friend.

function. A named group of statements that can be invoked and evaluated and can return a value to the calling statement.

function definition. The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

function prolog. The code that appears at the beginning of a function and that links stack frames, saves registers, and allocates automatic storage.

G

global. Pertaining to information available to more than one program or subroutine.

global variable. A variable defined in one portion of a computer program and used in at least one other portion of the computer program.

guard page. The page of memory allocated directly below the committed portion of the stack.

H

header file. A file that contains system-defined control information that precedes user data.

hexadecimal. A base sixteen numbering system; hexadecimal digits range from 0 through 9 (decimal 0 to nine) and uppercase or lowercase A through F (decimal ten to fifteen).

I

I/O Stream library. A class library that provides the facilities to deal with many varieties of input and output.

identifier. A sequence of letters, digits and underscores used to designate a data object or function.

IEEE. Institute of Electrical and Electronics Engineers.

include file. A file included with a, #include directive (#include).

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

incomplete class declaration. A class declaration that does not define any members of a class. Typically an incomplete class declaration is used as a forward declaration.

initialize. To set the starting value of a data object.

initializer. The assignment operator followed by an expression (or multiple expressions, for aggregate variables) used to set the initial value of a data object.

inlined function. A function call that the compiler replaces with the actual code for the function. You can direct the compiler to inline a function with the _Inline keyword and the /Oi compiler option.

input. Data to be processed.

input stream. A sequence of control statements and data submitted to a system from an input unit.

instance. Synonym for object, a particular instantiation of a data type.

instantiate. To create or generate a particular instance or object of a data type.

Institute of Electrical and Electronics Engineers (IEEE). A professional society that sponsors many standards activities, including the binary floating point standard sponsored by its Computer Society.

integer constant. A decimal, octal, or hexadecimal constant.

integral object. A character object, an object having an enumeration type, an object having variations of the type int, or an object that is a bit field.

intermediate code linker. A part of the C/C++ Tools compiler that combines the information in all intermediate code files to improve optimization.

internal data definition. A description of a variable appearing at the beginning of a block that causes storage to be allocated for the lifetime of the block.

interrupt. A temporary suspension of a process caused by an external event, performed in such a way that the process can be resumed.

intrinsic function. A function supplied by a program as opposed to a function supplied by the compiler.

K

keyword. (1) A predefined word reserved for the C or C++ language, that may not be used as an identifier. (2) A symbol that identifies a parameter.

L

label. (1) An identifier followed by a colon. It is the target of a goto statement. (2) An identifier within or attached to a set of data elements.

lexically. Relating to the left-to-right order of units.

library. (1) A collection of functions, function calls, subroutines, or other data. (2) A set of object modules that can be specified in a link command.

link. To interconnect items of data or portions of one or more computer programs; for example, linking of object programs by a linkage editor to produce an executable file.

linkage editor. Synonym for linker.

linker. A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single executable program.

local. Pertaining to information that is defined and available in only one function of a computer program.

long constant. An integer constant followed by the letter L in uppercase or lowercase.

lvalue. An expression that represents a data object that can be both examined and altered.

M

macro. An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor #define directive.

main function. A function with the identifier main that is the first user function to get control when program execution begins. Each C program must have exactly one function named main.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. The linker uses these mangled names to ensure type-safe linkage.

map. A set of values having a defined correspondence with the quantities or values of another set.

map file. A listing file that can be created during the link step and that contains information on the size and mapping of segments and symbols.

mapping. The establishing of correspondences between a given logical structure and a given physical structure.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

member. (1) A data object in a structure or a union. (2) In C++, classes and structures can also contain functions and types as members.

member function. An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class.

method. Synonym for member function.

migrate. To move to a changed operating environment, usually to a new release or version of a system.

module. A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

multibyte character. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multitasking. A mode of operation that allows concurrent performance, or interleaved execution of more than one task or program.

multithread. Pertaining to concurrent operation of more than one path of execution within a computer.

N

NaN. Not-a-Number.

nested class. A class defined within the scope of another class.

new. (1) A C++ keyword identifying a free storage allocation operator. (2) A C++ operator used to create class objects.

new-line character. A control character that causes the print or display position to move to the first position on the next line. This control character is represented by \n in the C language.

Not-a-Number (NaN). A binary bit value for a floating-point type that is not equal to any other valid floating-point value, including itself. A NaN is typically the result of an operation that is not valid, such as division of zero by zero. A NaN can be either a signalling NaN (NaNS) that raises signals or exceptions, or a quiet NaN (NaNQ) that does not.

NMAKE. A compiling and linking aid that searches for files changed since the last compilation and recompiles only the changed files.

NPX. Numeric processor extension.

NULL. A pointer guaranteed not to point to a data object.

null character (\0). The ASCII or EBCDIC character with the hex value (all bits turned off).

null value. A parameter position for which no value is specified.

O

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as C language).

object module. A portion of an object program produced by a compiler from a source program, and suitable as input to a linkage editor.

| **object-oriented programming**. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished, but on what data objects comprise the problem and how they are manipulated.

operand. An entity on which an operation is performed.

operating system. Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operation. A specific action such as add, multiply, shift.

operator. A symbol (such as +, -,) that represents an operation (in this case, addition, subtraction, multiplication).

| **operator function**. An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type.

OS/2. Pertaining to the operating system for the PS/2 workstation.

overflow. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overlay. To write over existing data in storage.

| **overloading**. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

pack. To store data in a compact form in such a way that the original form can be recovered.

pad. To fill unused positions in a field with data, usually zeros, ones, or blanks.

parameter declaration. A description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

parent process. The program that originates the creation of other processes by means of spawn or exec function calls. See also *child process*.

pointer. A variable that holds the address of a data object or function.

| **pointer to member**. An operator used to access the address of non-static members of a class.

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

precision. A measure of the ability to distinguish between nearly equal values.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

preprocessor statement. A statement that begins with the symbol # and is interpreted by the preprocessor.

primary expression. An identifier, a parenthesized expression, a function call, an array element specification, or a structure or union member specification.

private. Pertaining to a class member that is only accessible to member functions and friends of that class.

process. An instance of an executing application and the resources it uses.

program. One or more files containing a set of instructions conforming to a particular programming language syntax.

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters.

public. Pertaining to a class member that is accessible to all functions.

pure virtual function. A virtual function that has a function definition of = ;.

R

recoverable error. An error condition that allows continued execution of a program.

reentrant. The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

register. A storage area commonly associated with fast-access storage, capable of storing a specified amount of data such as a bit or an address.

reserved word. In programming languages, a keyword that may not be used as an identifier.

rounding. To omit one or more of the least significant digits in a positional representation and to adjust the remaining digits according to a specified rule. The purpose of rounding is usually to limit the precision of a number or to reduce the number of characters in the number.

run. To cause a program, utility, or other machine function to be performed.

runtime library. A collection of functions in object code form, whose members can be referred to by an application program during the linking step.

S

SAA. Systems Application Architecture.

scalar. An arithmetic object, or a pointer to an object of any type.

scope. That part of a source program in which an object is defined and recognized.

scope operator (::). An operator that defines the scope for the argument on the right. If the left argument is blank, the scope is global; if the left argument is a class name, the scope is within that class. Also called the scope resolution operator.

semaphore. An object used by multithread applications for signalling purposes and for controlling access to serially reusable resources.

signal. A condition that may or may not be reported during program execution. For example,