Targeting the NetWare® Operating System

PDK 5.0

Revised: 12/13/02
Metrowerks, the Metrowerks logo, and CodeWarrior are registered trademarks of Metrowerks Corp. in the US and/or other countries. All other tradenames and trademarks are the property of their respective owners.

© Copyright. 2002. Metrowerks Corp. ALL RIGHTS RESERVED.

The reproduction and use of this document and related materials are governed by a license agreement between Metrowerks Corp. and its licensee. Consult that license agreement before use or reproduction of any portion of this document. If you do not have a copy of the license agreement, contact your Metrowerks representative or call 800-377-5416.

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

Documentation stored on electronic media may be printed for non-commercial personal use only, further to the license agreement related to the product associated with the documentation. Subject to the foregoing non-commercial personal use, no portion of this documentation may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Metrowerks.

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

How to Contact Metrowerks:

| Voice USA: | 1-800-377-5416 |
| International: | +1-512-997-4700, Operator 4 |
| World Wide Web Metrowerks: | http://www.metrowerks.com/ |
| Motorola: | http://www.motorola.com/General/index.html |
# Table of Contents

## 1 Introduction
- Scope of Manual ........................................ 9
- Related Documentation .................................. 10
- Product Release Notes .................................. 11

## 2 Getting Started
- Installing the CodeWarrior Add-on Tools for NetWare .................................. 13
- Windows PC Requirements ................................ 14
- NetWare Server Requirements .............................. 14
- Installation Procedure ..................................... 15
- Overview: The NetWare Development Tools .................. 18
- The CodeWarrior IDE ...................................... 18
- The Makefile Importer Wizard ............................. 19
- Export Project as GNU Makefile .......................... 21
- The Project Manager ...................................... 22
- The Editor .................................................. 24
- The NetWare/x86 C/C++ Compiler ........................ 24
- The NetWare/x86 Linker .................................. 24
- The NetWare/x86 Debugger ............................... 25
- The NetWare/x86 Command Line Tools ................... 26
- The Metrowerks Standard Libraries ...................... 27
- The CodeWarrior Development Process .................. 28

## 3 Tutorial: Creating a NetWare Project
- NetWare Project Stationery ............................... 29
- Creating a NetWare Project ............................... 31
- Updating an Old Project .................................. 34

## 4 NetWare/x86 Target Settings
- Target Settings Overview ................................ 37
- General Target Settings Panels ......................... 39
- NetWare/x86-Specific Target Settings Panels ........... 39
  - Target Settings Panel .................................. 40
  - Runtime Settings Panel ............................... 43
  - Source Trees Panel ................................... 45
Access Paths Panel ........................................... 47  
NLM Target Panel ........................................... 47  
x86 CodeGen Panel ........................................... 50  
Mpxxdc Panel ................................................. 54  
NLM Linker Panel ........................................... 57  
x86 Disassembler Panel ....................................... 60  
Global Connection Preference Panel ....................... 63  
Required Files .............................................. 67  

5 The NetWare/x86 C/C++ Compiler  69  
Data Type Formats ........................................... 70  
 Integer Data Types ......................................... 70  
 Floating-Point Data Types ................................ 71  
Calling Conventions ........................................... 71  
 Supported Calling Conventions ............................. 72  
 Overriding the Default Calling Convention ............... 75  
Variable Allocation .......................................... 77  
Inlining Intrinsic Functions ................................ 77  
 Using the __builtin__ Prefix ................................. 79  
 Processor Port Intrinsic Functions ......................... 80  
Register Variables ........................................... 81  
Optimizing Code ............................................. 82  
 Register Coloring ........................................... 82  
 Instruction Scheduling ..................................... 84  
 Vectorization ................................................. 84  
 Loop Distribution ........................................... 87  
 Reduction Functions ......................................... 88  
Pragmas ......................................................... 89  
Inline Assembly Language .................................... 89  

6 80x86 Assembly Language Support  91  
Working with 80x86 Assembly Language ...................... 91  
Inline Assembly Language Syntax ............................ 92  
Directives ..................................................... 97  
Statement Labels ............................................ 99  
Comments ..................................................... 100
Preprocessor Support .................................................. 100
Valid Instruction Operands ......................................... 101
Local Variables and MMX Registers ......................... 102
Inlining Functions Containing asm Blocks .............. 103
Optimizing Inline Assembly Language ................... 103
Working with x86 Assembly Language for AMD Processors . 104

7 The NetWare/x86 Linker .................................................. 107
Linker Input Files ......................................................... 107
Object Modules .......................................................... 108
Static Libraries ........................................................ 108
NLM Files ................................................................. 108
Linker Command Files .............................................. 109
Symbol Export Files ................................................ 109
Symbol Import Files ................................................ 110
Link Order Files ........................................................ 112
Help Screen Files ...................................................... 112
Message Files ........................................................... 112
Remote Procedure Call Files .................................. 112
Linker Output Files ...................................................... 113
NetWare Loadable Modules ................................... 113
NLM Map Files .......................................................... 113
Dynamic Linking ......................................................... 120
Static Linking ............................................................ 120
Deadstripping ............................................................. 121
Link Order ................................................................. 121

8 Command Line Tools .................................................. 123
Conventions ................................................................. 123
Global Command Line Options ..................... 125
Compiler Options .......................................................... 128
Preprocessing, Precompiling, and Input File Control Options 128
C/C++ Language Options ....................................... 133
x86 Code Generation Options ....................... 139
Code Optimization Options ......................... 142
C/C++ Warning Options ....................................... 145
Debugging Control Options ............................ 148
Minimizing Mode Switching ........................................ 196
Pragmas ................................................................. 197
  Code Generation Pragmas ....................................... 197
  Constant Definition Pragmas .................................. 198
Name Mangling ....................................................... 199
Target Settings for AMD Processors .......................... 200
Athlon-Specific Assembly Language Instructions .......... 201
  Athlon-Specific 3DNow! Instructions ....................... 201
  Athlon-Specific MMX Instructions ......................... 202

A NetWare Linker Command Files ................................ 203
  Command File Format ......................................... 203
  Command Definitions ........................................... 204

Index ......................................................................... 223
Using the CodeWarrior™ Add-on Tools for NetWare®, you can develop applications, modules, and drivers for versions 3, 4, and 5 of the NetWare operating system. In addition, you can debug these programs using the PDK.

This manual explains features of the CodeWarrior Integrated Development Environment (IDE) that are specific to the CodeWarrior Add-on Tools for NetWare.

This chapter contains these sections:

- **Scope of Manual**
- **Related Documentation**
- **Product Release Notes**

**Scope of Manual**

This manual contains the following information:

- System requirements
- Installation instructions
- Descriptions of features specific to this product.
Related Documentation

For information outside the scope of this manual, consult the documents listed in Table 1.1 and Table 1.2.

Table 1.1 Related Metrowerks Documentation

<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior™ IDE User Guide</td>
<td>• Explains how to use features of the CodeWarrior IDE that are the same for all versions of the product.</td>
</tr>
</tbody>
</table>
| CodeWarrior™ C Compilers Reference | • Explains how to use the features of CodeWarrior C/C++ compiler.  
• Defines how the compiler implements the C and C++ languages.  
• Defines the compiler’s extensions to the C and C++ languages. |
| CodeWarrior™ MSL C++ Reference | • Describes each C++ class available in the C++ class library included in the Metrowerks Standard Libraries (MSL). |
| CodeWarrior™ Command Line Tools Reference | • Explains how to use the CodeWarrior command line C/C++ compiler, assembler, and linker. |

If you installed the CodeWarrior reference CD, the manuals can be accessed by selecting Help > Online Manuals from the main menu.

Table 1.2 Related NetWare Documentation

<table>
<thead>
<tr>
<th>Document</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novell Online Documentation</td>
<td>• <a href="http://developer.novell.com/ndk/doc.htm">http://developer.novell.com/ndk/doc.htm</a></td>
</tr>
<tr>
<td>NetWare ToolMaker Specification</td>
<td>• Explains how to use the NetWare ToolMaker product.</td>
</tr>
<tr>
<td>Mastering NetWare 5.1</td>
<td>• Third party book that explains how to develop software for NetWare 5.1.</td>
</tr>
</tbody>
</table>
Product Release Notes

Please read the release notes for the CodeWarrior Add-on Tools for NetWare.

The release notes file is named `Release notes.txt`. It is in the directory:

```
(Compiler)\Release Notes\AOT-NW5.0.notes
```

The release notes include:

- new features
- bug fixes
- last minute information after this manual was published
Getting Started

This chapter explains how to install the CodeWarrior™ PDK for NetWare®.

In addition, the chapter includes an overview of the NetWare development tools and development process.

This chapter contains these sections:
- Installing the CodeWarrior Add-on Tools for NetWare
- Overview: The NetWare Development Tools
- The CodeWarrior Development Process

Installing the CodeWarrior Add-on Tools for NetWare

You can install and use the CodeWarrior Add-on Tools for NetWare on a standalone Windows® PC.

To use the PDK’s integrated debugger, however, you must connect the PC to a NetWare server.

The following sections explain how to configure the equipment so that you can develop NetWare software and use the integrated debugger.

This section contains these topics:
- Windows PC Requirements
- NetWare Server Requirements
- Installation Procedure
Windows PC Requirements

The requirements to install and run the CodeWarrior Add-on Tools for NetWare on a Windows PC are listed in Table 2.1.

Table 2.1  System Requirements for Windows PC

<table>
<thead>
<tr>
<th>Component</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>An IBM-compatible PC.</td>
</tr>
<tr>
<td>Processor</td>
<td>80486, minimum. A Pentium-class processor is recommended.</td>
</tr>
<tr>
<td>Memory</td>
<td>64 MB, minimum.</td>
</tr>
<tr>
<td>Network Adapter Card</td>
<td>Any Ethernet card.</td>
</tr>
<tr>
<td>Serial Communications Port</td>
<td>One unused serial communications port.</td>
</tr>
<tr>
<td>Free disk space</td>
<td>33 MB.</td>
</tr>
<tr>
<td>Operating system</td>
<td>Windows NT SP6, Windows 2000 SP2, or Windows XP Professional.</td>
</tr>
<tr>
<td>NetWare NDK</td>
<td>Latest version available.</td>
</tr>
</tbody>
</table>

NetWare Server Requirements

The requirements to debug software on a NetWare server are listed in Table 2.2.

Table 2.2  System Requirements for NetWare Server

<table>
<thead>
<tr>
<th>Component</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>An IBM-compatible PC.</td>
</tr>
<tr>
<td>Processor</td>
<td>A Pentium-class processor.</td>
</tr>
<tr>
<td>Memory</td>
<td>128 MB, minimum.</td>
</tr>
</tbody>
</table>
Getting Started
Installing the CodeWarrior Add-on Tools for NetWare

Targeting the NetWare® Operating System, PDK 5.0

Installation Procedure

NOTE
Before you install the CodeWarrior Add-on Tools for NetWare, you must install the NetWare Development Kit (NDK) on the C: drive.

If you do not install the NDK as instructed, the NetWare project stationery will not work.

To install the CodeWarrior Add-on Tools for NetWare, follow these steps:

1. Place the CodeWarrior Add-on Tools for NetWare installation CD in the CD drive of your Windows PC.

The first screen of the installation program appears. See Figure 2.1.

<table>
<thead>
<tr>
<th>Component</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Adapter Card</td>
<td>One NE2000 compatible Ethernet card for normal LAN traffic.</td>
</tr>
<tr>
<td></td>
<td>One NE2000 compatible Ethernet card for debugging.</td>
</tr>
<tr>
<td></td>
<td>NOTE: The second card is required if you want to use the debugger.</td>
</tr>
<tr>
<td>Serial Communications Port</td>
<td>One unused serial communications port.</td>
</tr>
<tr>
<td></td>
<td>NOTE: This port is required if you want to debug using a serial communications link.</td>
</tr>
<tr>
<td>Operating System</td>
<td>NetWare 5.1 SP5 or NetWare 6 SP2</td>
</tr>
</tbody>
</table>
2. Select the type of installation you want from the options presented by the installation program.

To do this, follow these steps:

a. Read the instructions on each screen of the installation program.

b. Select the desired installation options presented on each screen.

c. Move between screens using the Back and Next buttons.

3. Click Finish on the last screen of the installation program.

4. Define the NovellNDK environment variable.

To do this, follow these steps:

a. Run the Window’s Settings control panel

b. Double-click System to display the System Properties dialog box
c. Click the Environment tab (Win98/NT4) of the System Properties dialog box, or in Win2000 click the Advanced tab and click Environment Variables button.

The Environment page of the System Properties dialog box appears. Figure 2.2 shows this dialog box.

![The System Properties Dialog Box](image)

- **Figure 2.2 The System Properties Dialog Box**

```plaintext
d. Click New... in the User Variables pane of the Environment Variables window.

e. Enter the string **NovellNDK** in the Variable name: text field.

f. Enter **C:\Novell\ndk\libc** in the Variable value: text field.

g. Click OK.

The values entered for Variable name and value appear in a new row of the User Variables pane.
```
h. Click OK to close window or dialog box.
Windows® creates the NovellNDK environment variable.
i. Log out of Windows and log back in again
The NovellNDK environment variable is available.

Overview: The NetWare Development Tools

This section discusses the development tools included in the CodeWarrior Add-on Tools for NetWare.

The topics in this section are:

• The CodeWarrior IDE
• The Makefile Importer Wizard
• Export Project as GNU Makefile
• The Project Manager
• The Editor
• The NetWare/x86 C/C++ Compiler
• The NetWare/x86 Linker
• The NetWare/x86 Debugger
• The NetWare/x86 Command Line Tools
• The Metrowerks Standard Libraries

The CodeWarrior IDE

The CodeWarrior IDE (Integrated Development Environment) is a program that provides a set of tools you use to develop NetWare programs.

The IDE has a graphical user interface (GUI). You use the GUI to control the tools included in the CodeWarrior Add-on Tools for NetWare.

The most important development tools provided by the IDE are the project manager, editor, compiler, linker, and debugger.

For complete documentation of the CodeWarrior IDE, refer to online help or the CodeWarrior™ IDE User Guide.
The Makefile Importer Wizard

The Windows-hosted CodeWarrior IDE can import a GNU make file and create a CodeWarrior project file. Use the makefile importer wizard to parse and convert make files into projects. Source files and build targets are identified and added to the project.

To import a makefile:
1. Click File > New...
2. Select Project tab
3. Select Makefile Importer Wizard (Figure 2.3)

4. Enter a project name in Project name: field
5. Click Set... button to locate folder for new project
   a. In Create New Project... dialog box, navigate to desired folder
   b. Click (check) Create Folder in lower left corner to create new folder with same name as project
   c. Click Save
6. Click OK – Makefile Importer Wizard dialog box appears

**Figure 2.4 Makefile Importer Wizard (Page 1)**

7. Enter path to makefile in Makefile Location field or click Browse to navigate to makefile

8. Select linker and tool set used for makefile conversion
   a. Tool Set Used In Makefile – In dropdown list, select tool set whose build rules form the basis of makefile. Select Standard Unix MAKE from the list that also includes Microsoft Visual C++ and Metrowerks Command Line.

   b. Metrowerks Tool Set – Select the linker tool set to use with generated project. Select NLM Linker.
9. Select (check) desired diagnostic settings – information is logged in a Project Messages window that appears in IDE
   a. Log Targets Bypassed – Logs information about makefile build targets that IDE fails to convert to project build targets
   b. Log Build Rules Discarded – Logs information about makefile rules that IDE discards during conversion
   c. Log All Statements Bypassed – Logs targets that were bypassed, build rules that were discarded, and other makefile items that IDE fails to convert

10. Click Finish – Summary of targets to be created appears

11. Click Generate – Makefile Importer wizard performs conversion process and displays new project window.

### Export Project as GNU Makefile

CodeWarrior NLM projects can be exported to a GNU makefile. The makefile (.mk) will be placed in your project’s output directory.

To export to makefile:

1. Open project
2. Select Project > Export Project as GNU Makefile – makefile will be saved in project output directory with .mk extension

The Project Manager

A project is a collection of related files and configuration settings that the CodeWarrior IDE uses to organize your software projects.

The project manager is a window that displays the files and targets your project uses.

Table 2.3 defines several project-related terms.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host</td>
<td>The system on which you run the CodeWarrior IDE to develop software for one or more platform targets.</td>
</tr>
<tr>
<td>Platform target</td>
<td>The operating system or microprocessor for which you develop software. The platform target can be different from the host.</td>
</tr>
<tr>
<td>Build target</td>
<td>A named collection of settings and files that the IDE uses to build a final output file. A build target defines all build-specific information, including: • Information that identifies files that belong to the build target • Compiler and linker settings for the build target • Output information for the build target</td>
</tr>
</tbody>
</table>

A project can contain multiple build targets. This allows you to define custom builds for different purposes.

The project manager keeps track of dependencies between files in your project. As a result, when you change a file and then build your project, the IDE compiles:
• the file you changed
• all files that are dependent on the file you changed
The project manager lets you define one or more build targets for the same project. A build target is a named set of project settings and files that the IDE uses to build a final output file.

For example, you could create a build target named Debug. For this target, you might choose settings that include information needed by the debugger.

Within the same project, you could also create a second build target, named Release. For this build target, you could exclude all debugging information so the release version of your program is smaller.

Figure 2.6 shows a project containing a debug build target and a release build target.

Figure 2.6   A Project with Multiple Build Targets

For complete documentation of the CodeWarrior project manager, refer to online help or the CodeWarrior™ IDE User Guide.
The Editor

The CodeWarrior IDE includes a text editor that includes many features useful to programmers.

For example, the editor highlights language keywords in the color you choose, interfaces with your source code control software, and more.

For complete documentation of the CodeWarrior editor, refer to online help or the CodeWarrior™ IDE User Guide.

The NetWare/x86 C/C++ Compiler

The CodeWarrior C/C++ compiler for NetWare/x86 is an ANSI-compliant compiler.

The compiler is part of the CodeWarrior IDE. The IDE invokes the compiler if you:

- Change a source file and issue the make command.
- Select a source file in your project and issue the compile, preprocess, or precompile command.

Table 2.4 indicates where to find more information about the compiler.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Source of Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information specific to the NetWare/x86 compiler.</td>
<td>• NetWare/x86-Specific Target Settings Panels.</td>
</tr>
<tr>
<td></td>
<td>• The NetWare/x86 C/C++ Compiler.</td>
</tr>
<tr>
<td>Inline assembly language support.</td>
<td>“80x86 Assembly Language Support” on page 91.</td>
</tr>
</tbody>
</table>

The NetWare/x86 Linker

The CodeWarrior linker for NetWare/x86 generates NetWare Loadable Modules™ (NLMs) by combining object modules created by the compiler and contained in static libraries.
The linker can create several types of output files for the NetWare/x86 environment. For example, the linker can create operating system NLMs, shared library NLMs, and LAN driver NLMs.

You define the type of output file the linker generates when you configure your project.

The linker is part of the CodeWarrior IDE. The IDE runs the linker each time you build your project.

Table 2.5 indicates where to find more information about the NetWare/x86 linker.

### Table 2.5  CodeWarrior Linker Documentation for NetWare/x86

<table>
<thead>
<tr>
<th>Topic</th>
<th>Source of Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linker input files</td>
<td>[Linker Input Files]</td>
</tr>
<tr>
<td>Linker output files</td>
<td>[Linker Output Files]</td>
</tr>
<tr>
<td>Other linker information</td>
<td>[The NetWare/x86 Linker]</td>
</tr>
</tbody>
</table>

### The NetWare/x86 Debugger

The CodeWarrior debugger for NetWare/x86 is one of the development tools available in the CodeWarrior IDE.

The debugger lets you execute your program’s statements one at a time and halt your program’s execution at a particular statement.

When your program is halted, you can examine and change the values of variables, view the contents of memory and registers, write to memory and registers, view the chain of function calls, and more.

The NetWare/x86 debugger also has these capabilities:

- Multi-threaded debugging
  - You can pause and resume execution of any thread in a multi-threaded NLM.
- Multi-processor debugging
You can control thread execution on multiple processors and view information for the processor on which the target thread is running.

- Multiple NLM debugging.

You can follow execution from one NLM to another. As you move from one NLM to the next, the debugger displays the source code for the current NLM.

Table 2.6 shows where to find more information about the debugger.

Table 2.6  NetWare/x86 Debugger Documentation

<table>
<thead>
<tr>
<th>Topic</th>
<th>Source of Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>General debugger features</td>
<td>The CodeWarrior™ IDE User Guide</td>
</tr>
<tr>
<td>Features specific to the NetWare/x86 debugger</td>
<td>The NetWare/x86 Debugger</td>
</tr>
</tbody>
</table>

The NetWare/x86 Command Line Tools

CodeWarrior Add-on Tools for NetWare includes command line versions of the NetWare/x86 C/C++ compiler and NetWare/x86 linker.

The PDK also includes a command line 80x86 assembler.

Table 2.7 lists the names of the command line compiler, linker, and assembler.

Table 2.7  Command Line Tool Names

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mwccnlm.exe</td>
<td>NetWare/x86 C/C++ compiler</td>
</tr>
<tr>
<td>mwldnlm.exe</td>
<td>NetWare/x86 linker</td>
</tr>
<tr>
<td>mwasmnlm.exe</td>
<td>NetWare/x86 assembler</td>
</tr>
</tbody>
</table>

These files are in the directory:

..\CodeWarrior\Other Metrowerks Tools\Command Line Tools
Both the IDE and command line versions of the linker accept object modules created by the command line compiler and the command line assembler.

Table 2.8 indicates where to find more information about the command line tools.

### Table 2.8 Documentation of the Command Line Tools

<table>
<thead>
<tr>
<th>Topic</th>
<th>Source of Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Information pertaining to all versions of the CodeWarrior command line tools.</td>
<td>The CodeWarrior™ Command-Line Tools Reference</td>
</tr>
<tr>
<td>Parameters for the NetWare/x86 command line compiler</td>
<td>Type mwccnlm -help at the command line</td>
</tr>
<tr>
<td>Parameters for the NetWare/x86 command line linker.</td>
<td>Type mwldnlm -help at the command line</td>
</tr>
<tr>
<td>Parameters for the NetWare/x86 command line assembler.</td>
<td>Type mwasmnlm -help at the command line</td>
</tr>
</tbody>
</table>

### The Metrowerks Standard Libraries

The CodeWarrior Add-on Tools for NetWare includes a set of ANSI-compliant C++ support libraries called the Metrowerks Standard Libraries (MSL).

Depending on your project, you must include some or all of these support libraries.

NetWare/x86 Support Libraries documents the requirements for using MSL.
The CodeWarrior Development Process

When using the CodeWarrior Add-on Tools for NetWare, you will often follow a process like that shown in Figure 2.7.

Figure 2.7  The CodeWarrior NetWare Development Process

- Start
  - Prepare to write source code
    - Write (edit) source code
      - Compile code
        - Link
          - Debug program
            - Release program
              - Finish

1. Connect a NetWare server to a Windows workstation.
2. Install the CodeWarrior PDK for NetWare on the workstation.
3. Create a project using project stationery.
4. Adjust target settings for the project, if necessary.
Tutorial: Creating a NetWare Project

This chapter consists primarily of a tutorial that shows you how to use the CodeWarrior™ PDK for NetWare® to create a new NetWare project.

The chapter starts by discussing project stationery because stationery is the tool you use most often to create a new project.

The chapter ends with a procedure that shows how to update projects created with stationery from earlier versions of the PDK.

This chapter contains these sections:

- NetWare Project Stationery
- Creating a NetWare Project
- Updating an Old Project

NetWare Project Stationery

Project stationery is a template that defines the files and project settings required to create a particular kind of NetWare Loadable Module™ (NLM).

If you use project stationery to create a new project, the CodeWarrior IDE creates a new project that is a duplicate of the selected stationery.

Table 3.1 lists the NetWare project stationery that you should use for new development.
### Table 3.1 Current NetWare Project Stationery (NKS-LibC)

<table>
<thead>
<tr>
<th>Project Stationery</th>
<th>Use to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traditional LibApp</td>
<td>Create a project for an NLM that uses (consumes) symbols exported by the NLM created with a Library project, which is autoloaded and consumed in the traditional NetWare manner.</td>
</tr>
<tr>
<td>Modern LibApp</td>
<td>Create a project for an NLM that uses (consumes) symbols exported by the NLM created with a Library project, using LoadLibrary.</td>
</tr>
<tr>
<td>Library</td>
<td>Create a project for a shared library NLM that uses both DLLMain and traditional means (get/set_app_data) to support NLM application consumers.</td>
</tr>
<tr>
<td>UI with NUT</td>
<td>Create a project for a user interface with a NUT NLM.</td>
</tr>
</tbody>
</table>

Table 3.2 lists legacy NetWare project stationery. This stationery is included in the PDK for downward compatibility with earlier releases.

### Table 3.2 Legacy NetWare Project Stationery (CLib)

<table>
<thead>
<tr>
<th>Project Stationery</th>
<th>Use to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic NLM</td>
<td>Create a project for a generic NLM.</td>
</tr>
<tr>
<td>Generic NLM C</td>
<td>Create a project for a generic NLM that will be written in the C language.</td>
</tr>
<tr>
<td>Generic NLM C++</td>
<td>Create a project for a generic NLM that will be written in the C/C++ language.</td>
</tr>
<tr>
<td>NDS - Add Object</td>
<td>Create a project for an NLM that adds an object to the NetWare Directory Service.</td>
</tr>
<tr>
<td>NDS - Delete Object</td>
<td>Create a project for an NLM that deletes an object from the NetWare Directory Service.</td>
</tr>
<tr>
<td>NDS - List Objects</td>
<td>Create a project for an NLM that lists an object in the NetWare Directory Service.</td>
</tr>
<tr>
<td>NDS - Modify Values</td>
<td>Create a project for an NLM that modifies values in the NetWare Directory Service.</td>
</tr>
<tr>
<td>Schema - Define New Attribute</td>
<td>Create a project for an NLM that defines a new schema attribute.</td>
</tr>
</tbody>
</table>
Creating a NetWare Project

To create a NetWare project, follow these steps:

1. Run the CodeWarrior IDE.
2. Select File > New.

The IDE displays the New dialog box. See Figure 3.1.

<table>
<thead>
<tr>
<th>Project Stationery</th>
<th>Use to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema - Define New Class</td>
<td>Create a project for an NLM that defines a new schema class.</td>
</tr>
<tr>
<td>Schema - Modify Class</td>
<td>Create a project for an NLM that modifies a schema class.</td>
</tr>
<tr>
<td>Shared Library NLM</td>
<td>Create a project for a shared library NLM.</td>
</tr>
<tr>
<td>Static Library NLM</td>
<td>Create a project for a static library.</td>
</tr>
<tr>
<td>User Interface - NUT</td>
<td>Create a project for a user interface NLM.</td>
</tr>
</tbody>
</table>
3. Click the Project tab in the New dialog box.

4. Select NetWare Stationery.

5. Enter the name of your new project in the Project name: text box. For this tutorial, enter the name **MyNLM**.

6. Enter the directory in which to create your new project in the Location: text box.
   
   You can type this directory name directly into the Location: text box or you can press the Set... button.
   
   The Set... button displays the Create New Project... dialog box. You can use this dialog box to select the desired directory.
   
   For this tutorial, enter **C:\NetWare_Projects\MyNLM**.

7. Click OK.

   The IDE displays the New Project dialog box. This dialog box is shown in **Figure 3.2**.
8. Select the NetWare project stationery for the type of project you must create.

Table 3.2 lists and describes the available NetWare project stationery.

For this tutorial, select the Generic C project stationery.

9. Click OK.

The CodeWarrior IDE creates a new project using the selected stationery (Figure 3.3).

The IDE names the project MyNLM.mcp and places it in the directory C:\Netware_Projects\MyNLM.
Refer to online help or the *CodeWarrior™ IDE User Guide* for instructions that explain how to:

- Add build targets to a project
- Add files to a build target
- Edit files
- Compile and link a build target
- Debug a build target

In addition, refer to *The NetWare/x86 Debugger* for an explanation of debugger features unique to the NetWare PDK.

Refer to the NetWare NDK documentation for instructions on how to develop software for the NetWare operating system.

## Updating an Old Project

The current version of the PDK lets you work with projects created with earlier versions of the PDK. You can also update an old project to take advantage of improvements made to the PDK.

To update an old project, follow these steps:

1. Run the CodeWarrior IDE
2. Open the old project
   The project window for the project appears.

3. Delete prelude.obj from the project window

**NOTE** This file might not be present, depending on the stationery used to create the project.

4. Remove clib.imp from the project window
5. Add libc.imp to the project window
6. Add libcpre.o to the project window
7. Select Edit > targetname Settings... (ALT-F7) to display the Target Settings window. See Figure 3.4.

**NOTE** The word Target does not appear in the title of the Target Settings window. Instead, the name of the project's current build target appears.
8. Select NLM Linker in the Target Settings Panel window. The IDE displays the NLM Linker settings panel.

9. In the Start: text box, replace _Prelude with _LibCPrelude.

10. In the Exit: text box, replace _Stop with _LibCPostlude.

11. Click OK to save your changes.
NetWare/x86 Target Settings

This chapter defines the NetWare-specific target settings you must make if you create a NetWare/x86 project starting from an empty project.

The chapter also defines the files that you must include in a NetWare/x86 project.

NOTE

The project stationery included in the CodeWarrior™ PDK for NetWare® sets all target settings correctly for NetWare/x86 development.

You need the information in this chapter only if you create a new NetWare/x86 project starting from an empty project.

This chapter contains these sections:

- Target Settings Overview
- General Target Settings Panels
- NetWare/x86-Specific Target Settings Panels
- Required Files

Target Settings Overview

Each build target in a CodeWarrior project has its own target settings.

You define target settings using settings panels available in the Target Settings window.
To display the target settings window:

1. Start the CodeWarrior IDE.
2. Select Edit > targetname Settings... (ALT-F7)

The Target Settings window appears. See Figure 4.1.

In this figure, the current build target is named Generic C Build and shows the C/C++ Language panel.

![Figure 4.1 The Target Settings Window for the Generic C Build Target](image-url)
General Target Settings Panels

Some panels are for general CodeWarrior build target settings. Other panels require NetWare/x86-specific target settings.

Table 4.1 lists each target settings panel that is not NetWare/x86-specific. The table also lists the CodeWarrior manual that documents each of these panels.

### Table 4.1 General CodeWarrior Target Settings Panels

<table>
<thead>
<tr>
<th>Target Settings Panel</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build Extras</td>
<td>CodeWarrior™ IDE User Guide</td>
</tr>
<tr>
<td>File Mappings</td>
<td>CodeWarrior™ IDE User Guide</td>
</tr>
<tr>
<td>C/C++ Language</td>
<td>CodeWarrior™ C Compilers Reference</td>
</tr>
<tr>
<td>C/C++ Warnings</td>
<td>CodeWarrior™ C Compilers Reference</td>
</tr>
<tr>
<td>Global Optimizations</td>
<td>CodeWarrior™ IDE User Guide</td>
</tr>
<tr>
<td>Custom Keywords</td>
<td>CodeWarrior™ IDE User Guide</td>
</tr>
<tr>
<td>Other Executables</td>
<td>CodeWarrior™ IDE User Guide</td>
</tr>
<tr>
<td>Debugger Settings</td>
<td>CodeWarrior™ IDE User Guide</td>
</tr>
<tr>
<td>Remote Debugging</td>
<td>CodeWarrior™ IDE User Guide</td>
</tr>
<tr>
<td></td>
<td>NOTE: this panel is not used for NetWare/x86 development</td>
</tr>
</tbody>
</table>

NetWare/x86-Specific Target Settings Panels

The panels addressed in this section are:

- Target Settings Panel
- Runtime Settings Panel
- Source Trees Panel
- Access Paths Panel
- NLM Target Panel
- x86 CodeGen Panel
- Mpkxdc Panel
- NLM Linker Panel
NetWare/x86 Target Settings
NetWare/x86-Specific Target Settings Panels

- x86 Disassembler Panel
- Global Connection Preference Panel

These panels require NetWare/x86-specific settings.

Target Settings Panel

The Target Settings panel is in the Target group of panels. Use this panel to specify your platform target. Windows® and NetWare are examples of platform targets.

NOTE
The Target Settings window contains a Target Settings panel. The window and the panel are not the same.
The window displays one settings panel at a time.

Figure 4.2 shows the Target Settings panel.
Figure 4.2 Target Settings Panel

Table 4.2 lists the entries you must make in the Target Settings panel for NetWare/x86 development.

Table 4.2 NetWare/x86-Specific Target Settings Panel Items

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Required Entry/Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Name:</td>
<td>Name of the build target</td>
</tr>
<tr>
<td>Linker:</td>
<td>NLM Linker</td>
</tr>
</tbody>
</table>

Targeting the NetWare® Operating System, PDK 5.0
NetWare/x86 Target Settings
NetWare/x86-Specific Target Settings Panels

The default values for items not listed in Table 4.2 are acceptable for NetWare/x86 projects.

### File Mappings Panel

To add the .xdc file extension:

1. Open project’s Target Settings Panels (Alt-F7)
2. Select File Mappings panel
3. Delete text in File Type box
4. Enter .xdc in Extension: box
5. Choose None from Compiler: dropdown list
6. Click Add and OK buttons

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Required Entry/Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-linker:</td>
<td>None or Pre Linker MPKXDC Runner</td>
</tr>
<tr>
<td></td>
<td>The Pre Linker plugin is used to execute mpkxdc.exe, which is a Novell utility that runs before linking and produces an output file: filename.xdc. The linker uses this .xdc file in the link process to build an NLM.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> To use this feature, ensure that the .xdc file extension exists in the File Mappings Panel.</td>
</tr>
<tr>
<td>Post-linker:</td>
<td>None or NLM Post Linker</td>
</tr>
<tr>
<td></td>
<td>NLM Post Linker is used when copying a NLM file to a specific directory on the NetWare server.</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> This feature requires user to install the NetWare Client for Windows and a second NE2000 compatible Ethernet card in the server for LAN communication.</td>
</tr>
</tbody>
</table>
Figure 4.3  File Mappings Panel

Runtime Settings Panel

Use the Runtime Settings panel to configure the environment your program needs when it runs under control of the IDE or debugger.

Figure 4.4 shows the Runtime Settings panel. This panel contains items used for NetWare/x86 development.
Table 4.3 lists and explains the entries you can make in the Runtime Settings panel.
Table 4.3 Runtime Settings Panel Items

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host Application for Libraries and Code Resources</td>
<td>Enter the name of the executable that loads the libraries and code resources you want to debug.</td>
</tr>
<tr>
<td>General Settings</td>
<td></td>
</tr>
<tr>
<td>• Working Directory</td>
<td>Name of working directory your program uses while running under the control of the IDE.</td>
</tr>
<tr>
<td>• Program Arguments</td>
<td>Command line arguments passed to your program each time it runs under control of the IDE.</td>
</tr>
<tr>
<td>Environment Settings</td>
<td>Enter environment variables your program needs when run under the control of the IDE.</td>
</tr>
</tbody>
</table>

Source Trees Panel

Use the Source Trees panel to assign unique names to absolute paths, environment variables, and registry keys. These names are called *root paths*.

You can use a root path name when defining an access path. See Access Paths Panel for an explanation of access paths.

Figure 4.5 shows the Source Trees panel.
Table 4.4 lists the entries you must make in the Source Trees panel for NetWare/x86 development.

Table 4.4  NetWare/x86-Specific Source Tree Panel Entries

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Required Entry/Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name:</td>
<td>NDK</td>
</tr>
<tr>
<td>Type:</td>
<td>Environment Variable</td>
</tr>
<tr>
<td>Environment Variable Name</td>
<td>NovellNDK</td>
</tr>
</tbody>
</table>
Access Paths Panel

Use the Access Paths panel to define search paths.

The compiler and linker use these paths to find files whose location is not clearly specified, such as header files and libraries.

Figure 4.6 shows the Access Paths panel.

Figure 4.6 Access Paths Panel

For instructions explaining how to use the Access Paths panel, refer to online help or the CodeWarrior™ IDE User Guide.

NLM Target Panel

Use the NLM Target panel to define the type of NLM your project creates.
Figure 4.7 shows the NLM Target panel.

**Figure 4.7  The NLM Target Panel**

The NLM Target panel is only used for NetWare/x86 development. Therefore, this section explains all items in this panel.

Table 4.5 lists and explains the entries you can make in the NLM Target settings panel.

**Table 4.5  NLM Target Panel Items**

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output Type:</td>
<td>Select the linker output file type. The options are listed in the bulleted rows that follow.</td>
</tr>
<tr>
<td>• Generic NLM (.NLM)</td>
<td>General network loadable module</td>
</tr>
<tr>
<td>• LAN Driver (.LAN)</td>
<td>Network interface card driver</td>
</tr>
</tbody>
</table>
### NetWare/x86 Target Settings

#### NetWare/x86-Specific Target Settings Panels

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Disk Driver (.DSK)</td>
<td>Disk driver module</td>
</tr>
<tr>
<td>• Name Space Support (.NAM)</td>
<td>Name space support application</td>
</tr>
<tr>
<td>• Utility (.NLM)</td>
<td>Utility module</td>
</tr>
<tr>
<td>• Mirrored Server Link (.MSL)</td>
<td>Mirrored server link module</td>
</tr>
<tr>
<td>• OS NLM (.NLM)</td>
<td>Operating system network loadable module</td>
</tr>
<tr>
<td>• Paged High OS NLM (.NLM)</td>
<td>Paged high operating system network loadable module</td>
</tr>
<tr>
<td>• Host Adapter (.HAM)</td>
<td>Host adapter module</td>
</tr>
<tr>
<td>• Custom Device (.CDM)</td>
<td>Custom device module</td>
</tr>
<tr>
<td>• Object Library (.LIB)</td>
<td>Static library</td>
</tr>
</tbody>
</table>

**Output File:** Enter the name of the output file

**Screen Name:** Specify the screen your NLM uses to display output. The options are explained in the bulleted rows that follow.

- • User Specified
  The NLM uses the screen you specify in the Initial Screen Name item (see below) for output.

- • Console
  The NLM uses the console’s screen for output.

- • No Default
  The NLM uses no output screen.

**Initial Screen Name:**

Enter the name of the screen that your NLM uses for output.

**NOTE:** This panel item is disabled unless you select User Specified for the Screen Name: item.

**Stack Size:**

Enter the stack size for your NLM.

**NOTE:** the minimum stack size is 2048.

You can use the `STACK` or `STACKSIZE` commands in a linker command file instead of this panel item.

**Initial Thread Name:**

Enter the prefix to use in the name of each thread created by your NLM.

You can use the `THREADNAME` command in a linker command file instead of this panel item.
**Panel Item** | **Description**
--- | ---
Copyright: | Enter the copyright text for your NLM. This text displays on the console or user specified screen when the NLM loads.

You can use the COPYRIGHT command in a linker command file in place of this panel item.

Description: | Enter any text desired.

NOTE: Your NLM will not link unless you enter something for this panel item, or include a DESCRIPTION command in linker command file.

Version Major: | Enter the major version number for your NLM.

This item accepts only numeric characters.

Version Minor: | Enter the minor version number for your NLM.

This item accepts only numeric characters.

Version Revision: | Enter the revision version number for your NLM.

Enter numbers 0 through 26 for this item.

Enter 0 to specify no revision number.

A number between 1 and 26 corresponds to the letter of the alphabet, for example 1=A and 26=Z.

You can use the VERSION command in a linker command file in place of the Major, Minor, and Revision panel items.

---

**x86 CodeGen Panel**

Use the x86 CodeGen panel to control the type of code the compiler generates.

**Figure 4.8** shows the x86 CodeGen panel.
Use the x86 CodeGen panel for NetWare/x86 development. This section explains all items in this panel.

Table 4.6 lists and explains the entries you can make in the x86 CodeGen panel.
### Table 4.6  x86 CodeGen Panel Items

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime configuration:</td>
<td>Select the target runtime library configuration. All selections (except Custom) define _MT, _MSL_IMP_EXP, _DLL, _AFXDLL, and _DEBUG for use in configuring MSL and MFC.</td>
</tr>
<tr>
<td>Target processor:</td>
<td>Select the x86 instruction set for the compiler to use. Your choices are listed in the bulleted rows that follow.</td>
</tr>
<tr>
<td>• Generic 80x86</td>
<td>The compiler uses just those instructions that execute on Intel® 80386, 80486, all Pentium™ processors and processors compatible with these processors.</td>
</tr>
<tr>
<td>• Pentium</td>
<td>The compiler generates instructions appropriate for the selected processor.</td>
</tr>
<tr>
<td>• Pentium Pro</td>
<td>The compiler does not use any instructions specific to the selected processor unless you enable one or more of the options in the Extended Instructions group (see below).</td>
</tr>
<tr>
<td>• Pentium II</td>
<td></td>
</tr>
<tr>
<td>• Pentium III</td>
<td></td>
</tr>
<tr>
<td>• Pentium IV</td>
<td></td>
</tr>
<tr>
<td>• AMD K6</td>
<td>The compiler generates MMX™ multimedia instructions. This option is available for all processor types (except Generic 80x86).</td>
</tr>
<tr>
<td>• AMD Athlon</td>
<td>The compiler generates 3DNow!™ multimedia instructions. This option is available for AMD processors.</td>
</tr>
<tr>
<td>• AMD Athlon XP</td>
<td></td>
</tr>
<tr>
<td>Processor-specific</td>
<td>Select the extended instruction set(s) that the compiler can use. The choices are explained in the bulleted rows that follow.</td>
</tr>
<tr>
<td>Instructions:</td>
<td></td>
</tr>
<tr>
<td>• MMX</td>
<td>The compiler generates MMX™ multimedia instructions. This option is available for all processor types (except Generic 80x86).</td>
</tr>
<tr>
<td>• 3DNow!</td>
<td>The compiler generates 3DNow!™ multimedia instructions. This option is available for AMD processors.</td>
</tr>
<tr>
<td>• Conditional moves</td>
<td>The compiler generates conditional move instructions, for example (a ? b : c) constructs.</td>
</tr>
<tr>
<td></td>
<td>This option is available for the Pentium Pro, Pentium II / III / IV processors, and AMD Athlon processors.</td>
</tr>
<tr>
<td>Panel Item</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SSE</td>
<td>Allows the compiler to handle inline assembly language SSE instructions for single-precision floating-point code generation and vectorization. This option is available for the Pentium III and IV processors and AMD Athlon XP processor.</td>
</tr>
<tr>
<td>SSE-2</td>
<td>Allows the compiler to handle inline assembly language SSE-2 instructions for double-precision floating-point code generation and vectorization on the Pentium IV processor.</td>
</tr>
<tr>
<td>Struct alignment:</td>
<td>Specify how the compiler should align the boundary for structure and class members. Each member aligns on a multiple of either the specified alignment or size of the member, whichever is greater.</td>
</tr>
<tr>
<td></td>
<td>NOTE: Select 1 for all NetWare/x86 development.</td>
</tr>
<tr>
<td>Inline intrinsic</td>
<td>Enable this option if you want the compiler to inline the intrinsic functions shown in Listing 5.3.</td>
</tr>
<tr>
<td>functions</td>
<td>NOTE: This feature is not part of the ANSI standard.</td>
</tr>
<tr>
<td>Optimize inline</td>
<td>Enable this item to schedule and peephole inline assembly code.</td>
</tr>
<tr>
<td>assembly</td>
<td></td>
</tr>
<tr>
<td>Disable optimizations for debug</td>
<td>For easier debugging, enable this item to turn off some optimizations. For example, by always generating a stack frame.</td>
</tr>
<tr>
<td>C++ exception</td>
<td>Specify how the compiler should handle processor exceptions. When disabled, try...catch tables are generated for Metrowerks Zero-overhead exceptions. When enabled, generate code and data compatible with synchronous exceptions used in MSVC++.</td>
</tr>
<tr>
<td>handling</td>
<td>NOTE: Disable this item for all NetWare/x86 development.</td>
</tr>
<tr>
<td>Name mangling</td>
<td>When disabled, supports the full C++ standard as implemented in the CodeWarrior IDE. When enabled, does not support CodeWarrior extensions and is bug-compatible with MSVC++ 6.0.</td>
</tr>
<tr>
<td></td>
<td>NOTE: Disable this item for all NetWare/x86 development.</td>
</tr>
</tbody>
</table>
Mpkxdc Panel

Use the Mpkxdc panel to define settings for the Novell Mpkxdc utility and specify a remote target directory on the Netware server in which to download your NLM programs.

Figure 4.9 shows the Mpkxdc panel.

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Multimedia Calling Convention</td>
<td>Enable this option to speed the execution of AMD’s 3DNow! multimedia extensions.</td>
</tr>
<tr>
<td></td>
<td>NOTE: Enable this option only if you have also enabled the 3DNow! instruction set.</td>
</tr>
<tr>
<td></td>
<td>To learn more about optimizing for the AMD K6 and Athlon processors, see Targeting AMD Processors.</td>
</tr>
<tr>
<td>Make strings read-only</td>
<td>Enable this item to place literal strings into the read-only data (.rdata) section instead of the read-write (.data) section.</td>
</tr>
</tbody>
</table>
Use the Mpkxdc panel for NetWare/x86 development. This section explains all items in this panel.

Table 4.7 lists and explains the entries you can make in the Mpkxdc panel.
## Table 4.7 Mpkxdc Panel Items

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Switches</strong></td>
<td>Enter command-line options for the Mpkxdc utility. Available switches:</td>
</tr>
<tr>
<td>-n</td>
<td>Generate XDC data for an MT safe NLM. (mpkxdc -n OutputFile)</td>
</tr>
<tr>
<td>-f</td>
<td>Generate XDC data for funnelling APIs listed in InputFile (mpkxdc -f InputFile OutputFile)</td>
</tr>
<tr>
<td>-b</td>
<td>Generate v4.11 SMP compatible XDC data for NOT funnelling any API (mpkxdc -b OutputFile)</td>
</tr>
<tr>
<td>-d</td>
<td>Generate v4.11 SMP compatible XDC data for NOT funnelling APIs listed in InputFile (mpkxdc -d InputFile OuputFile)</td>
</tr>
<tr>
<td>-u</td>
<td>Generate XDC data to declare an NLM to be MT unsafe (mpkxdc -u OutputFile)</td>
</tr>
<tr>
<td>-p</td>
<td>Generate XDC data to mark the NLM preemptible</td>
</tr>
</tbody>
</table>

**NOTE:** For more information on this utility, refer to the Impk1.txt file located in the CodeWarrior_Install_Path\Novell Support\Mpkxdc folder or execute "mpkxdc -h" at a command prompt.

| **API files** | If required, an input file that contains a list of APIs. The file must list one API per line without leading or trailing white spaces or trailing commas. |
| **Output file** | File name to which generated XDC data will be written. It is the same file name you specify in the Output File box in the NLM Target panel. |
| **Target Output Directory** | Type or browse to path of target folder where NLM files are copied for debugging on the NetWare server. |

**NOTE:** When you build (make) your NLM project in the CodeWarrior IDE, the NLM file is downloaded to the path specified. This feature requires you to install the NetWare Client for Windows and a second NE2000 compatible Ethernet card in the server for LAN communication.
NLM Linker Panel

Use the NLM Linker panel to control how the linker generates its output file.

Figure 4.10 shows the NLM Linker panel.

Figure 4.10  NLM Linker Panel

Use the NLM Linker panel for NetWare/x86 development. This section explains all items in this panel.

Table 4.8 lists and explains the entries you can make in the NLM Linker panel.
## Table 4.8  NLM Linker Panel Items

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry Points</td>
<td>Enter names of entry points for routines that perform the functions in the bulleted rows that follow.</td>
</tr>
<tr>
<td>• Start:</td>
<td>Enter name of routine that contains startup code required by an NLM.</td>
</tr>
<tr>
<td></td>
<td>If you are using <code>libc.imp</code>, enter <code>_LibCPrelude</code>.</td>
</tr>
<tr>
<td></td>
<td>If you are using <code>clib.imp</code>, enter <code>_Prelude</code>.</td>
</tr>
<tr>
<td></td>
<td>You can use the <strong>START</strong> command in a linker command file instead of this panel item.</td>
</tr>
<tr>
<td>• Exit:</td>
<td>Enter name of routine that contains the cleanup code required by an NLM.</td>
</tr>
<tr>
<td></td>
<td>If you are using <code>libc.imp</code>, enter <code>_LibCPostlude</code>.</td>
</tr>
<tr>
<td></td>
<td>If you are using <code>clib.imp</code>, enter <code>_Stop</code>.</td>
</tr>
<tr>
<td></td>
<td>You can use the <strong>EXIT</strong> command in a linker command file in place of this panel item.</td>
</tr>
<tr>
<td>• Check:</td>
<td>Enter name of routine in your NLM that you want executed when the NetWare console operator uses the <strong>UNLOAD</strong> console command.</td>
</tr>
<tr>
<td></td>
<td>This function must return 0 if your NLM can be unloaded. The function must return a non-zero value if your NLM cannot be unloaded.</td>
</tr>
<tr>
<td></td>
<td>You can use the <strong>CHECK</strong> command in a linker command file in place of this panel item.</td>
</tr>
<tr>
<td>Generate Link Map</td>
<td>Enable this option to generate a link map file.</td>
</tr>
<tr>
<td>Generate CV SYM File</td>
<td>Enable this option to generate a CodeView symbol file. The CodeWarrior debugger requires this file.</td>
</tr>
<tr>
<td>Allocate Uninitialized Data in File</td>
<td>Enable this option to cause the linker to allocate memory for static and global variables and initialize this memory to zero.</td>
</tr>
<tr>
<td></td>
<td>Disable this option to let the runtime library allocate and initialize memory for global and static variables.</td>
</tr>
<tr>
<td>Panel Item</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| Generate Internal Debugger Records | Enable this option to include debugger information in the output file.  
The NetWare debugger, RDEBUG, requires this information.  
The CodeWarrior debugger does not require this information.  
You can use the DEBUG command in a linker command file in place of this panel item. |
| Flags                            | Click the Flags button to display the flags menu. Select each flag that you want the linker to turn on in your NLM.  
The available flags are described in the bulleted rows that follow. |
| • Reentrant                      | When enabled (checked), allows multiple threads to execute the NLM’s code simultaneously.  
You can use the REENTRANT command in a linker command file in place of this flag. |
| • Multiple                       | When enabled, allows NetWare to load the NLM more than once.  
You can use the MULTIPLE command in a linker command file in place of this flag. |
| • Synchronize                    | When enabled, causes the load process to sleep until the NLM calls SynchronizeStart(). This prevents other console commands from being processed while the NLM is loading.  
You can use the SYNCHRONIZE command in a linker command file in place of this flag. |
| • PseudoPreemption               | When enabled, allows NetWare to force the NLM to relinquish the processor if the NLM does not do so often enough on its own.  
You can use the PSEUDOPREEMPTION command in a linker command file in place of this flag. |
NetWare/x86 Target Settings
NetWare/x86-Specific Target Settings Panels

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>• OS Domain</td>
<td>When enabled, causes the NLM to be loaded in the operating system’s domain.</td>
</tr>
<tr>
<td></td>
<td>You can use the <strong>OS_DOMAIN</strong> command in a linker command file in place of this flag.</td>
</tr>
<tr>
<td>• Auto Unload</td>
<td>When enabled, causes NetWare to unload the NLM when its entry points are no longer in use.</td>
</tr>
<tr>
<td></td>
<td>You can use the <strong>AUTOUNLOAD</strong> command in a linker command file in place of this flag.</td>
</tr>
</tbody>
</table>

x86 Disassembler Panel

Use the x86 Disassembler panel to control what is displayed in the disassembly window.

You can display disassembled code using the debugger. For information about the debugger, see *The NetWare/x86 Debugger* and refer to online help or the *CodeWarrior™ IDE User Guide*.

*Figure 4.11* shows the x86 Disassembler target settings panel.
Figure 4.11  The x86 Disassembler Panel

Use the x86 Disassembler panel for NetWare/x86 development. This section explains all items in the panel.

Table 4.9 lists and explains the entries you can make in the x86 Disassembler panel.

Table 4.9  x86 Disassembler Panel Items

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object headers</td>
<td>Enable this option to include header information in a disassembly.</td>
</tr>
<tr>
<td>Section headers</td>
<td>Display section header table</td>
</tr>
</tbody>
</table>
### Panel Item | Description
---|---
Symbol and string tables | Enable this option to include this information in a disassembly:
- A symbol table
- A string table
- Each symbol at its point of definition

Code sections | Enable this option to include text sections in a disassembly. When this option is enabled, the additional options described in the bulleted rows that follow are available.

- Interleave source | The disassembly window is organized such that each line of source code is followed by the assembly language instructions generated for that line. For this feature to work, these things must be true:
  - The disassembled file must have been built with debugging information.
  - The IDE must be able to find the source code used to produce the disassembled file.

- Show address and opcodes | The disassembly shows the address of each instruction and bytes defining the instruction.

- Show comments | The disassembly includes comments produced by the disassembler in sections where the comment columns are provided.

- Resolve local references | Stack locations and registers will be replaced with names of locals. Enable Show comments to see the original memory location or register.

Data sections | Enable to include data sections in disassembly.

Debug sections | Enable to use debugging information in a disassembly.

If enabled, named locals and arguments appear in a disassembly, and the original operand appears in the comment field.

NOTE: If this option is used with highly optimized code, registers that are colored to locals or arguments are also given names. This can be misleading because such a register might not always represent the particular local or argument named because the register can be used for multiple purposes.
Global Connection Preference Panel

Use the Global Connection preference panel to set up the communications link used by the Windows PC and the NetWare server during a debugging session.

NOTE

The Global Connection panel is not a target settings panel. It is an IDE preference panel in the Debugger group of the IDE Preferences window (Edit > Preferences...).

You can choose between two types of debugger communications links: serial or TCP/IP.

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception tables</td>
<td>Enable to include exception tables in a disassembly. The disassembly displays Metrowerks zero-overhead exception tables in human-readable format. Microsoft-compatible exception tables are displayed in hex format.</td>
</tr>
<tr>
<td>Relocation tables</td>
<td>Enable to include relocation tables in a disassembly.</td>
</tr>
<tr>
<td>Display hex dumps</td>
<td>Enable to include unstructured hexadecimal dumps in a disassembly.</td>
</tr>
<tr>
<td>Dump undecoded sections</td>
<td>If enabled, remaining undecoded sections are displayed as hex dumps.</td>
</tr>
<tr>
<td>Resolve relocations</td>
<td>The disassembly includes relocation information for the corresponding text or data section. The disassembly also shows the names of symbols whose addresses will be relocated.</td>
</tr>
<tr>
<td>Show symbol definitions</td>
<td>If enabled, labels will be inserted into the disassembly at the point where symbols are defined.</td>
</tr>
<tr>
<td>Unmangle symbols</td>
<td>If selected, symbols are unmangled whenever they appear. Otherwise, symbols are unmangled only at their definitions.</td>
</tr>
<tr>
<td>Verbose Information</td>
<td>Enable to include additional information in a disassembly, such as auxiliary symbols, section header flags, and unsupported fields.</td>
</tr>
</tbody>
</table>
If you choose View Serial Settings from the View Connection Type: dropdown menu, the Global Connection panel displays items you can use to set up a serial communications link.

If you choose View TCP/IP Settings, the panel displays items you can use to set up a TCP/IP communications link.

Figure 4.12 shows the Global Connection panel with View Serial Settings selected.

Table 4.10 lists and explains the entries you can make in the Global Connection panel for a serial communications link.

Figure 4.12 The Global Connection Panel Showing Serial Items
### Table 4.10  Global Connection Panel Items for a Serial Link

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>View Connection Type:</td>
<td>Select View Serial Settings to display the serial connection panel items.</td>
</tr>
<tr>
<td>Port:</td>
<td>Select the serial port for the debugger to use.</td>
</tr>
<tr>
<td></td>
<td>NOTE: For NetWare debugging, you must select COM1, COM2, COM3, or COM4.</td>
</tr>
<tr>
<td>Rate:</td>
<td>Select the baud rate at which the debugger and the NetWare server communicate.</td>
</tr>
<tr>
<td></td>
<td>NOTE: baud rates 300, 1200, and 230400 bits per second are not supported.</td>
</tr>
<tr>
<td>Data Bits:</td>
<td>Select the number of data bits in each character sent by the debugger.</td>
</tr>
<tr>
<td></td>
<td>NOTE: For NetWare debugging, you must select 8 data bits.</td>
</tr>
<tr>
<td>Parity:</td>
<td>Select the type of parity to use for each character sent by the debugger.</td>
</tr>
<tr>
<td></td>
<td>NOTE: For NetWare debugging, you must select none.</td>
</tr>
<tr>
<td>Stop Bits:</td>
<td>Select the number of stop bits for each character sent by the debugger.</td>
</tr>
<tr>
<td></td>
<td>NOTE: For NetWare debugging, you must select 1 stop bit.</td>
</tr>
<tr>
<td>Flow Control:</td>
<td>Select the type of flow control for the debugger to use.</td>
</tr>
<tr>
<td></td>
<td>NOTE: For NetWare debugging, you must select none.</td>
</tr>
<tr>
<td>Log Serial Data to Log Window</td>
<td>The debugger displays its output in a log window.</td>
</tr>
</tbody>
</table>

Figure 4.13 shows the panel with View TCP/IP Settings selected from the View Connection Type: dropdown menu.

Table 4.11 lists and explains the entries you can make in the Global Connection panel for a TCP/IP communications link.
Figure 4.13  The Global Connection Panel Showing TCP/IP Items

Table 4.11  Global Connection Panel Items for a TCP/IP Link

<table>
<thead>
<tr>
<th>Panel Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>View Connection Type:</td>
<td>Select View TCP/IP Settings to display the TCP/IP connection panel items.</td>
</tr>
<tr>
<td>Host Name:</td>
<td>Enter the domain name or IP address of the NetWare server Ethernet card used for debugging.</td>
</tr>
</tbody>
</table>
Required Files

All NetWare/x86 projects (except those that produce a static library) must include either the release or debug version of the Metrowerks C runtime library.

The names of these libraries are listed in Table 4.12.

Table 4.12 Runtime Library Required by all NetWare/x86 Projects

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mwcrt1.lib</td>
<td>Non-debug version of the C runtime library.</td>
</tr>
<tr>
<td>mwcrtld.lib</td>
<td>Debug version of the C runtime library.</td>
</tr>
</tbody>
</table>

No other files are required by all NetWare/x86 projects. However, a particular NetWare project type might require additional files.
The NetWare/x86 C/C++ Compiler

This chapter discusses features unique to the NetWare®/x86 implementation of the CodeWarrior™ C/C++ compiler.

Table 5.1 shows where to find documentation for general features of the CodeWarrior C/C++ compiler.

### Table 5.1 Additional Compiler Documentation

<table>
<thead>
<tr>
<th>For information on this topic...</th>
<th>See this manual...</th>
</tr>
</thead>
<tbody>
<tr>
<td>How the CodeWarrior IDE implements the C/C++ language</td>
<td>The CodeWarrior™ C Compilers Reference</td>
</tr>
<tr>
<td>Using the C/C++ Language and C/C++ Warnings settings panels</td>
<td>The CodeWarrior™ C Compilers Reference, “Setting C/C++ Compiler Options” chapter</td>
</tr>
<tr>
<td>Controlling the size of C++ code</td>
<td>The CodeWarrior™ C Compilers Reference, “C++ and Embedded Systems” chapter</td>
</tr>
<tr>
<td>Using compiler pragmas</td>
<td>The CodeWarrior™ C Compilers Reference, “Pragmas and Symbols” chapter</td>
</tr>
<tr>
<td>Initiating a build, controlling which files are compiled, and handling error reports</td>
<td>The CodeWarrior™ IDE User Guide, “Compiling and Linking” chapter</td>
</tr>
<tr>
<td>Information about error messages</td>
<td>The CodeWarrior™ Error Reference</td>
</tr>
</tbody>
</table>

This chapter contains these sections:

- [Data Type Formats](#)
- [Calling Conventions](#)
- [Variable Allocation](#)
Data Type Formats

This section describes how the NetWare/x86 C/C++ compiler implements integer and floating-point data types for 80x86 processors.

See limits.h for more information about integer types.

See float.h for more information about floating-point types.

Integer Data Types

The NetWare/x86 compiler does not let you change the size of integers:

- The size of a short int is always 2 bytes.
- The size of an int or long int is always 4 bytes.

Table 5.2 shows the size and range of integer data types for the NetWare/x86 C/C++ compiler.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Option Setting</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>n/a</td>
<td>8 bits</td>
<td>true or false</td>
</tr>
<tr>
<td>char</td>
<td>Use Unsigned Chars is off</td>
<td>8 bits</td>
<td>-128 to 127</td>
</tr>
<tr>
<td></td>
<td>Use Unsigned Chars is on</td>
<td>8 bits</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>n/a</td>
<td>8 bits</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>n/a</td>
<td>8 bits</td>
<td>0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>n/a</td>
<td>16 bits</td>
<td>-32,768 to 32,767</td>
</tr>
</tbody>
</table>
Floating-Point Data Types

Table 5.3 shows the sizes and ranges of floating-point data types supported by the NetWare/x86 C/C++ compiler.

Table 5.3 Floating Point Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Option Setting</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>n/a</td>
<td>32 bits</td>
<td>1.17549e-38 to 3.40282e+38</td>
</tr>
<tr>
<td>short double</td>
<td>n/a</td>
<td>64 bits</td>
<td>2.22507e-308 to 1.79769e+308</td>
</tr>
<tr>
<td>double</td>
<td>n/a</td>
<td>64 bits</td>
<td>2.22507e-308 to 1.79769e+308</td>
</tr>
<tr>
<td>long double</td>
<td>n/a</td>
<td>64 bits</td>
<td>2.22507e-308 to 1.79769e+308</td>
</tr>
</tbody>
</table>

Calling Conventions

This section discusses the function calling conventions that the NetWare/x86 C/C++ compiler supports.

This section contains these topics:

- Supported Calling Conventions
- Overriding the Default Calling Convention
Supported Calling Conventions

Table 5.4 lists the supported calling conventions.

Table 5.4  Supported Function Calling Conventions

<table>
<thead>
<tr>
<th>Calling Convention</th>
<th>DeclarationSpecifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>__cdecl</td>
</tr>
<tr>
<td>Pascal</td>
<td>__stdcall</td>
</tr>
<tr>
<td>C++</td>
<td>None.</td>
</tr>
<tr>
<td></td>
<td>The compiler uses this convention as needed. You cannot instruct the compiler to use it.</td>
</tr>
<tr>
<td>Fast call</td>
<td>__fastcall</td>
</tr>
</tbody>
</table>

The default behavior of the compiler is to use:

- The C calling convention for module-level functions.
- The C++ calling convention for structure and class member functions.

For instructions explaining how to override the compiler’s default calling convention behavior, see “Overriding the Default Calling Convention” on page 75.

The C language calling convention

When the C calling convention is in effect, the calling function:

- Pushes the called function’s arguments on the stack in right to left order.
  
  As a result, the left-most argument in the called function’s argument list is on top of the stack when control transfers to the called function.

- Expands each of the called function’s arguments to at least 32 bits of stack space.

- Pads structures used as arguments to an even number of 32 bit long words of stack space.

- Removes the arguments from the stack after the called function returns.
A function often returns a value to its caller. However, the called function returns this value differently depending on the data type of the returned value.

When the C calling convention is in effect, the possibilities are:

- **Integer and pointer return types.**
  For a 32-bit return type, the called function returns the value in the EAX register.
  For a `long long` return type, the called function returns the value in the EDX:EAX registers.

- **Floating point return types.**
  The called function places the returned value on the floating point unit’s (FPU) stack.

- **4 byte structure or class return types.**
  The called function returns the structure in the EAX register.

- **8 byte structure or class return types.**
  The called function returns the structure in the EDX:EAX registers.

- **For larger structures and classes:**
  - The calling function pushes the address of a temporary argument on the stack after it pushes all of the called function’s explicit arguments.
  - Upon return, the called function puts the address of the structure or class in this temporary argument.

**NOTE**

The compiler uses the C calling convention for all module level functions unless you instruct it to do otherwise.

For instructions explaining how to use the other calling conventions, see "Overriding the Default Calling Convention" on page 75.

`__cdecl` is the declaration specifier for the C calling convention.
The Pascal calling convention

The Pascal calling convention is the same as the C calling convention with one difference: Before it returns, the called function removes the arguments pushed on the stack by the calling function.

__stdcall is the declaration specifier for the Pascal calling convention.

The C++ language calling convention

The C++ language calling convention is the same as the C language calling convention, with one difference: In addition to doing everything the C calling convention does, the C++ calling convention passes the this pointer to the called function in the ECX register.

For every non-static member function declared in a class or structure, the compiler automatically uses the C++ calling convention.

NOTE

The compiler uses the C++ language calling convention for non-static member functions unless you instruct it to do otherwise.

For instructions explaining how to use the other calling conventions, see “Overriding the Default Calling Convention” on page 75.

There is no declaration specifier for the C++ calling convention. You can override the C++ calling convention, but not instruct the compiler to use it.

The fast call calling convention

The fast call calling convention reduces the time a called function needs to retrieve its arguments by placing two of these arguments in the ECX and EDX registers instead of on the stack.
The fast call calling convention places different arguments in ECX and EDX depending upon the type of the called function:

- C functions.
  The fast call calling convention places the first two arguments that are 4 bytes or smaller in the ECX and EDX registers.
- Non-static structure or class member functions that do not return a structure.
  The fast call calling convention places the this pointer in the ECX register and the first argument that is 4 bytes or smaller in the EDX register.
- Non-static structure or class member functions that return a structure.
  The fast call calling convention places the this pointer in the ECX register and a pointer to a temporary memory location in the EDX register. The called member function returns the structure to the location in the EDX register.

__fastcall is the declaration specifier for the fast call calling convention.

**Overriding the Default Calling Convention**

You can override the compiler’s default calling convention. You should do this if:

- The compiler’s default calling convention does not meet your needs.
  For example, one of your functions might be executed many times. In this case, the fast call convention might be suitable.
- Your program uses a function that does not use the compiler’s default calling convention.
  For example, your program might use a function in a third party library. If this library’s functions use the Pascal calling convention, you must instruct the compiler to call this library’s functions this way.

You can override the default calling convention for individual functions or for an entire module.
To override the default convention for an entire module, use the statement `#pragma calling_convention arg`, where `arg` can be:

- `__cdecl`
- `__stdcall`
- `__fastcall`
- `reset`

Listing 5.1 shows how to instruct the compiler to use the Pascal calling convention for an entire module.

**Listing 5.1  Overriding the Default Calling Convention for a Module**

```c
// Use the Pascal calling convention for this module
#pragma calling_convention __stdcall

// function prototypes
int myFunc1(void);
int myFunc2(void);

// function definitions
int myFunc1(void) {
    return 1;
}

int myFunc2(void) {
    return 1;
}
```

To override the compiler’s default calling convention for a specific function:

- Prefix the function’s prototype with one of the declaration specifiers listed in Table 5.4.
- Prefix the function’s definition with the same declaration specifier used in its prototype.

For example, to instruct the compiler to use the Pascal calling convention for the function `displayMsg()`, use the prototype and function definition shown in bold in Listing 5.2.
Listing 5.2  Overriding the Default Calling Convention for a Function

// function prototypes

// compiler uses the Pascal calling convention for displayMsg
int __stdcall displayMsg(char* msg);

// compiler uses the C calling convention for other functions
int        myFunc1(); // no need for __cdecl
float      myFunc2(); // no need for __cdecl

... // more module level statements

// definition of displayMsg
int __stdcall displayMsg(char* msg) {
   return( printf("\n%s", msg) );
}

Variable Allocation

The NetWare/x86 C/C++ compiler places no limits on how large variables can be or how you allocate them.

Inlining Intrinsic Functions

For some functions, the NetWare/x86 C/C++ compiler can insert the machine code that implements a function into the object module it is building each time it encounters a call to that function. This is called inlining an intrinsic function.

Normally, the compiler generates call/return machine code for these function calls, just as it does for other function calls.

The inline intrinsic function feature produces faster, but larger code.

NOTE  This feature is not part of the ANSI standard.
The functions for which the compiler can perform this action are called *intrinsic functions*. The prototypes for these functions are listed in *Listing 5.3*.

**Listing 5.3  Prototypes of Intrinsic Functions That Can Be Inlined**

```c
// memory allocation operations
char * _alloca(size_t size);
char * __alloca(size_t size);

// block memory operations
void * memchr(const void *src, int c, size_t n);
int memcmp(const void *s1, const void *s2, size_t n);
void * memcpy(void *dest, const void *src, size_t n);
void * memmove(void * dest, const void * src, size_t n);
void * memset(void * dest, int c, size_t n);

// wide character block memory operations
wchar_t * wmemchr(const wchar_t *, wchar_t, size_t);
int wmemcmp(const wchar_t *, const wchar_t *, size_t);
wchar_t * wmemcpy(wchar_t *, const wchar_t *, size_t);
wchar_t * wmemmove(wchar_t *, const wchar_t *, size_t);
wchar_t * wmemset(wchar_t *, wchar_t, size_t);

// string operations
int strcmp(const char *, const char *);
char * strcpy(char *, const char *);
size_t strlen(const char *);

// wide character string operations
int wcscmp(const wchar_t *, const wchar_t *);
wchar_t * wcscpy(wchar_t *, const wchar_t *);
size_t wcslen(const wchar_t *);

// floating point math operations
double sin(double x);
double cos(double x);
double tan(double x);
double atan(double x);
double atan2(double x);
double acos(double x);
double asin(double x);
```
double log10(double x);
double log(double x);
double exp(double x);
double pow(double x);
double sqrt(double x);
double fabs(double x);

// x86 processor port operations
unsigned char _inp(unsigned short port);
unsigned short _inpw(unsigned short port);
unsigned long _inpd(unsigned short port);
void _outp(unsigned short port, unsigned char val);
void _outpw(unsigned short port, unsigned short val);
void _outpd(unsigned short port, unsigned long val);

To use the inline intrinsic function feature:

1. Enable the Inline Intrinsic Functions option in the x86 CodeGen target settings panel.
   Refer to “x86 CodeGen Panel” on page 50 for an explanation of this panel item.

2. Prototype each intrinsic function you want the compiler to inline, in each source file in which you want the function inlined.
   You can do this in one of two ways:
   a. By including the function’s header file in each source file.
   b. By typing the function’s prototype directly into each source file.

If you do these things, the compiler inlines each intrinsic function it finds in the source files in which you included the intrinsic function prototypes.

**Using the __builtin__ Prefix**

You can instruct the compiler to inline a specific intrinsic function even if the Inline Intrinsic Function option is disabled.

To do this, follow these steps:

1. Include the __builtin__ prefix in the function’s prototype.
NOTE: You must enter this kind of prototype manually because the standard C header files do not include prototypes of the intrinsic functions that include the __builtin__ prefix.

2. Call the function by the name specified in the prototype. Listing 5.4 shows code that uses the __builtin__ prefix.

Listing 5.4 Using the __builtin__ Prefix

```c
#include <math.h> // prototypes of standard C math functions
double __builtin_log10(double x); // explicit function prototype

void myFunc() {
    double result;

    // function inlined no matter what
    result = __builtin_log10( 100 );

    // function inlined if Inline Intrinsic Functions option is on
    result = log10( 1000 );
}
```

Processor Port Intrinsic Functions

The processor port intrinsic functions manipulate the ports connected to an x86 processor. These functions are listed at the end of Listing 5.3.

The processor port intrinsic functions correspond to the IN and OUT instructions of the x86 processor family. There is a variant of each function for byte, word and double word operands.

NOTE: The processor port intrinsic functions may fail unless your application obtains proper port access privileges from NetWare.
Register Variables

This section describes how the NetWare/x86 C/C++ compiler allocates a function’s arguments and local variables to processor registers.

**NOTE**
The NetWare/x86 C/C++ compiler always ignores the `register` keyword. Instead, the compiler uses its own algorithm to determine which variables to place in processor registers.

The compiler automatically allocates a function’s arguments and local variables to registers based upon:

- How frequently the argument or local variable is used.
- How many processor registers are available.

The compiler will not allocate a function argument or local variable to a register if any statement within the function uses the address of the argument or variable.

If you optimize your program for execution speed, the compiler gives preference to local variables used inside loops.

The compiler also uses the floating point unit’s stack (ST(0) through ST(7)) for local floating-point variables.

The NetWare/x86 compiler can use these registers for integral-typed function arguments and local variables:

- EAX
- EBX
- ECX
- EDX
- ESI
- EDI

**NOTE**
The compiler can also use the EBP register for local variables at optimization level 1 or above. However, this can make debugging difficult.
To avoid this problem, you can use the pragma
#pragma use_frame on.

This pragma instructs the compiler to use the EBP register for a
stack frame instead of for a local variable.

## Optimizing Code

This section discusses code optimizations that are available only in
the NetWare/x86 implementation of the C/C++ compiler.

The Global Optimizations target settings panel provides additional
code optimization choices.

Refer to the CodeWarrior™ IDE User Guide for an explanation of each
optimization available in the Global Optimizations panel.

### NOTE

For optimization techniques specific to Advanced Micro Devices®
processors, refer to “Targeting AMD Processors” on page 195.

The NetWare/x86-specific optimizations are:

- Register Coloring
- Instruction Scheduling
- Vectorization
- Loop Distribution
- Reduction Functions

### Register Coloring

The NetWare/x86 C/C++ compiler can perform an optimization
called register coloring.

In this optimization, the compiler assigns two or more variables to
the same processor register. The compiler does this if a function’s
code does not use the variables at the same time.

In the example shown in Listing 5.5, the compiler could place \( i \) and
\( j \) in the same register.
Listing 5.5  Function Eligible for the Register Coloring Optimization

```c
int main(void) {
    short i, int j;

    for (i=0; i<100; i++)
        { MyFunc_1(i); }
    for (j=0; j<100; j++)
        { MyFunc_1(j); }
}
```

However, in the example in Listing 5.6, the compiler cannot place \(i\) and \(j\) in the same register because \(\text{main()}\) includes a statement that uses \(i\) and \(j\) at the same time. In this case, the compiler would place \(i\) and \(j\) in different registers.

Listing 5.6  Function Not Eligible for the Register Coloring Optimization

```c
int main(void) {
    short i, int j;

    for (i=0; i<100; i++)
        { MyFunc_1(i); }
    for (j=0; j<100; j++)
        { MyFunc_1(i + j); }  //compiler cannot use register coloring
}
```

Register coloring reduces code size and increases execution speed.

To enable the register coloring optimization, choose optimization Level 2 or higher in the Global Optimizations target settings panel.

To learn more about the Global Optimizations panel, refer to the *CodeWarrior™ IDE User Guide*.

**NOTE**  If you use the register coloring optimization to create a program that you later debug, it may look as if there is something wrong with local variables that share the same register.
In the example in Listing 5.5, the debugger would report that \( i \) and \( j \) always have the same value. Further, any value assigned to \( i \) would appear to also be assigned to \( j \) and vice versa.

Therefore, it is suggested that you debug only programs created with optimizations disabled.

### Instruction Scheduling

The compiler uses the *instruction scheduling* optimization to increase a program’s execution speed.

This optimization rearranges processor instructions so the execution of one instruction does not hold up the execution of others.

This optimization increases execution speed on processors that can execute more than one instruction at a time. These processors include Pentium, Athlon, and newer processors.

To enable the instruction scheduling optimization, choose Level 3 or Level 4 in the Global Optimizations target settings panel.

To learn more about the Global Optimizations panel, refer to the *CodeWarrior™ IDE User Guide*.

The processor you select in the x86 CodeGen target settings panel defines the scheduling heuristics the compiler uses for the instruction scheduling optimization.

For more information about the x86 CodeGen panel, see “x86 CodeGen Panel” on page 50.

### Vectorization

Vectorization is an optimization in which the compiler uses vector instructions to combine operations and their operands.

The NetWare/x86 compiler can perform this optimization, but only if your code conforms to certain rules. These rules are defined in the sections that follow.
Loop vectorization

When performing loop vectorization, the compiler inspects the operations and operands within a loop to decide whether it can use vector instructions to combine operations and operands.

If the compiler uses vector instructions, the processor completes more operations per cycle and executes fewer loop branches per iteration.

The compiler can use loop vectorization if a loop:

- Is a counted loop whose counter is incremented.
- Contains just assignment statements.
- Contains no control code or pointer references.

Loops that meet these criteria often perform operations on static or local arrays because the base address of such arrays is known at compile time.

Listing 5.7 shows the general form of a loop on which the compiler can perform loop vectorization.

Listing 5.7 General Form of a Loop That Can Be Vectorized

```c
for( i = initial; i op N; i = i + 1 ) {
    ... = a[ i + k1 ];
    a[ i + k ] = ...;
    ... = a[ i + k2 ];
    a[ i + k3 ] = ...;
}
```

In the example in Listing 5.7:

- `i` is the loop counter.
- `a` is a static array.
- `op` is either the `<` or `<=` operator.
- References on the right side of an assignment statement are loads from `a`.
- References on the left side of an assignment statement are stores to `a`. 
In general, the expression on the right side of an assignment can include:

- One or more vector operands
- Constants
- Loop invariant scalars
- Expressions and vector operations

The compiler can vectorize a loop that meets the restrictions listed in the next two sections.

**Loop restrictions**

A loop that can be vectorized must meet these restrictions:

- The loop exit condition must use either the < or the <= operator.
- The loop counter must have a unit stride.
- The loop counter must be used only in a unit stride dimension of an array access.
- The loop body must use the loop counter only in array index expressions.
- The loop counter must be used in array index expressions only within the innermost loop of a nested loop.
- The loop must be the innermost loop of a nested loop.
- There must be no flow of control statements in the loop.
- There must be no function calls in the loop.

**Dataflow restrictions**

A loop that can be vectorized must meet these data flow restrictions:

- There must be no loop carried data dependencies.

In the example in Listing 5.7, the loop cannot be vectorized if any of the following conditions existed:

- k > k1
- k2 > k
- k3 > k

The loops in Listing 5.8 cannot be vectorized because doing so would change their semantics.
Listing 5.8 Semantics Changed by Loop Vectorization

```c
//this loop violates the k > k1 condition
for( i = initial; i < N; i = i + 1 ) {
    b[ i ] = a[ i - 1 ];
    a[ i ] = b[ i ];
}
//this loop violates the k2 > k condition
for( i = initial; i < N; i = i + 1 ) {
    a[ i ] = b[ i ];
    a[ i + 1 ] = b[ i ];
}
//this loop violates the k3 > k condition
for( i = initial; i < N; i = i + 1 ) {
    a[ i ] = b[ i ];
    b[ i ] = a[ i + 1 ];
}
```

Vectorization requires that all loads and stores be unambiguous at compile time. Therefore, if a statement within a loop performs a load or store through a pointer, the compiler cannot vectorize the loop.

### Loop Distribution

Loop distribution is an optimization the NetWare C/C++ compiler can perform on loops that can be only partially vectorized.

An example of a loop that is a candidate for loop distribution is one that contains two statements, but one statement is not vectorizable. Listing 5.9 shows such a loop.

Listing 5.9 Loop That Is a Candidate for Loop Distribution

```c
for ( i = 0; i < 10; i++ ) {
    c[ i ] = b[ i ] * c[ i ]; // can be vectorized
    a[ i ] = b[ i ] * a[ i - 1 ]; // cannot be vectorized
}
```

In the case shown in Listing 5.9, the compiler splits the original loop into two loops: one loop is vectorized, the other is not.
Reduction Functions

A reduction function is a loop that contains an equation involving an assignment to a scalar value.

Listing 5.10 shows an example of a reduction function.

Listing 5.10  Example Reduction Function

```c
scalar = scalar OP vector
```

When most compilers try to employ vectorization to optimize loop code, they can usually vectorize only assignment statements whose LHS (left-hand side) is a vector. If the LHS of an assignment is a scalar, most compilers cannot vectorize the statement.

The NetWare/x86 C/C++ compiler, however, can vectorize loops that have a scalar on the LHS of an assignment statement.

For example, the code in Listing 5.11 can be vectorized by the NetWare/x86 compiler despite the fact that there is a scalar on the LHS of the assignment statement.

Listing 5.11  Reduction Functions on Scalars

```c
float a[100], b[100];
float scalar;

fred()
{
    scalar = 0.0f;

    for ( i = 0; i < 100; i++ ) {
        scalar = scalar + a[ i ] + b[ i ];
    }
}
```

For more on vectorization, see Vectorization
Pragmas

The pragmas supported by the NetWare/x86 C/C++ compiler are documented in the CodeWarrior™ C Compilers Reference manual.

Inline Assembly Language

The NetWare/x86 C/C++ compiler supports inline assembly language statements.

This feature is documented in 80x86 Assembly Language Support.
80x86 Assembly Language Support

The NetWare®/x86 C/C++ compiler includes an inline assembler. The inline assembler lets you embed assembly language statements in your C and C++ source code. This chapter explains the features of the inline assembler.

NOTE

The CodeWarrior™ PDK for NetWare® also includes a command line assembler.

For documentation of the command line assembler, refer to the CodeWarrior™ Assembler Reference manual.

This chapter contains these sections:

- Working with 80x86 Assembly Language
- Working with x86 Assembly Language for AMD Processors

Working with 80x86 Assembly Language

This section explains how to use the NetWare/x86 C/C++ compiler’s inline assembly language features.

NOTE

This chapter does not document the 80x86 instruction set.

Topics in this section consist of:

- Inline Assembly Language Syntax
- Directives
- Statement Labels
• Comments
• Preprocessor Support
• Valid Instruction Operands
• Local Variables and MMX Registers
• Inlining Functions Containing asm Blocks
• Optimizing Inline Assembly Language

Inline Assembly Language Syntax

This section defines the NetWare/x86 C/C++ compiler’s inline assembly language syntax.

This section contains these topics:
• The asm keyword
• Support for Microsoft inline assembly language
• General inline assembly language syntax rules
• Assembly language functions
• Assembly language statements within C and C++ functions
• Assembly language statements within macros

The asm keyword

To instruct the compiler to treat a series of statements as a block of assembly language statements, use the asm keyword.

The asm keyword is part of the ANSI C/C++ standard.

Rules for using the asm keyword:
• Enable the ANSI Strict option in the C/C++ Language target settings panel to assemble blocks of assembly language statements prefixed with the asm keyword.
• If you enable the ANSI Keywords Only option, blocks of assembly language statements prefixed with the asm keyword:
  – Assemble successfully if in a C++ source file.
  – Do not assemble successfully if in a C source file.
The ANSI Strict and ANSI Keywords Only options of the C/C++ Language target settings panel are explained in the CodeWarrior™ IDE User Guide and the CodeWarrior™ C Compilers Reference manual.

Support for Microsoft inline assembly language

The NetWare/x86 C/C++ compiler allows you to prefix blocks of assembly language instructions with the __asm keyword.

The __asm keyword is a Microsoft® extension to the ANSI C /C++ standard.

When you use the __asm keyword, assembly language instructions are always assembled, no matter how you set the ANSI Strict and ANSI Keywords Only options.

General inline assembly language syntax rules

All inline assembly language statements must follow these rules:

- Each statement must obey the syntax diagram shown in Listing 6.1.

Listing 6.1 Inline Assembly Language Syntax Diagram

```
[LocalLabel:] (instruction | directive) [operands]
```

- A single statement or block of statements must be preceded by the asm or __asm keyword. See Listing 6.2.

Listing 6.2 Use of the asm Keyword

```
asm mov eax, 0  //a single inline assembly language statement
__asm mov eax, 0  //another inline assembly language statement
asm {
    mov eax, 0
    mov ebx, 0
}
```
• Each statement on the same line must begin with the `asm` or `__asm` keyword. See Listing 6.3.

Listing 6.3  Multiple Statements on One Line

```asm
asm mov eax, 0 asm mov ebx, 1 // Correct
asm mov eax, 0 mov ebx, 1    // Incorrect
```

• A block of statements must be surrounded by braces `{ }`. See Listing 6.4.

Listing 6.4  A Block of Inline Assembly Language Statements

```asm
asm {
    mov eax, 0
    mov ebx, 1
}
```

• All hexadecimal constants must be expressed using ANSI C syntax or Microsoft® MASM/Visual C++ syntax. See Listing 6.5.

Listing 6.5  Hex Constants in Inline Assembly Language Statements

```asm
asm {
    mov eax, 0xABCDEF // OK - ANSI C-style hex constant
    mov ebx, 0ABCDEF  // OK - Microsoft-style hex constant
    mov ecx, $ABCDEF // ERROR - Pascal-style hex constant
}
```

• All binary constants must be expressed using Metrowerks syntax or Microsoft MASM/Visual C++ syntax. See Listing 6.6.

Listing 6.6  Binary Constants in Inline Assembly Language Statements

```asm
asm {
    mov eax, 0b10100 // OK - Metrowerks-style binary constant
    mov eax, 10100b  // OK - Microsoft-style binary constant
}
```
Assembly language functions

You can include functions written entirely in assembly language in your C and C++ source files.

To create an assembly language function, use the syntax shown in Listing 6.7. The keyword __declspec(naked) instructs the compiler not to generate function prologue and epilogue code.

Listing 6.7 Assembly Language Functions in C and C++ Source Files

```c
__declspec(naked) void myAssemblyLanguageFunc(void)
{
    asm {
        push     ebp
        mov      ebp,esp
        sub      esp,0x8
        mov      eax,0xcccccccc
        mov      dword ptr [esp],eax
        mov      dword ptr [esp+0x4],eax
        mov      dword ptr [ebp-0x8],0x1
        mov      dword ptr [ebp-0x4],0x10
        mov      eax,0x0
        leave
        ret      near
    } // end assembly language block
}
```

NOTE For the stack trace feature of the debugger to work for an inline assembly language function, you must set up a standard stack frame.

You do this by entering the function with the PUSH EBP and MOV EBP, ESP instructions and exiting the function with the LEAVE instruction or the MOV ESP, EBP and POP EBP instructions.

Assembly language statements within C and C++ functions

You can include an assembly language instruction or block of instructions within C and C++ functions.
Within a function, you can put an assembly language statement anywhere you can put a C/C++ statement.

Listing 6.8 shows a block of inline assembly instructions within a standard C function.

Listing 6.8 Inline Assembly Instructions within a Function

```c
int MyFunc(void)
{
    int i = 1;

    asm {
        mov eax, 0
        mov i, eax
    }

    return i; // i contains 0
}
```

NOTE Unless you instruct it to do so, the compiler does not optimize inline assembly language, whether it is in a C/C++ function or in a function consisting entirely of assembly language.

For C/C++ functions that contain inline assembly language, however, the compiler does optimize the function’s C/C++ code.

See Optimizing Inline Assembly Language for instructions that explain how to optimize blocks of inline assembly language statements.

Assembly language statements within macros

You can define a macro that includes an assembly language statement or statements.

However, because the preprocessor places an expanded macro on one line, all the assembly language statements in a macro are placed on the same line. Consequently, each assembly language statement in a macro must start with the keyword `asm`.
Listing 6.9 shows a macro definition that expands to produce incorrect code. The code in bold is incorrect because it does not begin with the `asm` keyword.

Listing 6.10 shows a macro definition that expands to produce correct inline assembly language code.

**Listing 6.9  An Incorrect Inline Assembly Language Macro**

```c
#define ASM_MACRO_BAD 
asm { 
   mov eax, 3 
   mov ebx, 4 
}

// ASM_MACRO_BAD expands to ...
asm { mov eax, 3 mov ebx, 4 } // ERROR: mov ebx, 4
   // must start with asm
```

**Listing 6.10  A Correct Inline Assembly Language Macro**

```c
#define ASM_MACRO_GOOD 
asm mov eax, 3 
asm mov ebx, 4

// ASM_MACRO_GOOD expands to ...
asm mov eax, 3 asm mov ebx, 4 //OK: mov ebx, 4 starts with asm
```

**Directives**

A **directive** is a keyword that instructs the inline assembler to do something.

The NetWare/x86 inline assembler supports the directives listed in Table 6.1.
Table 6.1  Directives Supported by the NetWare/x86 Inline Assembler

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>align</td>
<td>Aligns the next machine instruction on the next DWORD (double word) memory location.</td>
</tr>
<tr>
<td>align $n$</td>
<td>Aligns the next machine instruction on the next $n$-byte memory location.</td>
</tr>
<tr>
<td>align $n$, $m$</td>
<td>Aligns the next machine instruction on the next $n$-byte memory location, if this location is within $m$ bytes.</td>
</tr>
<tr>
<td>db $x$, $y$, ...</td>
<td>Inserts the specified value or values into the instruction stream. The compiler truncates the value specified to a value between 0 and 0xFF. This directive can also accept: * Expressions. * C-style single and multibyte characters ('*', 'ab', 'CWIE').</td>
</tr>
<tr>
<td>dw $x$, $y$, ...</td>
<td>Inserts the specified word value or values into the instruction stream. A word is 2-bytes long. The compiler truncates the value specified to a value between 0 and 0xFFFF. This directive can also accept: * Expressions. * C-style single and multibyte characters ('*', 'ab', 'CWIE').</td>
</tr>
<tr>
<td>dd $x$, $y$, ...</td>
<td>Inserts the specified double word value or values into the instruction stream. The compiler truncates the value specified to a value between 0 and 0xFFFFFFFF. This directive can also accept: * Expressions. * C-style single and multibyte characters ('*', 'ab', 'CWIE').</td>
</tr>
<tr>
<td>_emit byte</td>
<td>Inserts the specified byte value into the instruction stream.</td>
</tr>
</tbody>
</table>
Statement Labels

You can use a statement label to name a block of assembly language statements.

You can use a statement label as the target of a goto statement and jump instructions.

Rules for statement label names:

- A statement label name cannot be the same as the name of a local variable.
- A statement label name can start with the at-symbol (@).
- A statement label name must end in a colon.

The code shown in bold in Listing 6.11 shows statement labels.

**Listing 6.11  Statement Labels in Inline Assembly Language**

```c
void myFunc(void) {
    asm {
        x1:  mov eax, 2 // OK, has a terminating colon
        @x2: mov eax, 2 // OK, has both @ and colon
        x3  mov eax, 2 // ERROR: needs a terminating colon
    }
}
```

Instructs the compiler to optimize the inline assembly language instructions that follow.

You can turn off optimization within a block of assembly language statements using the nooptimize keyword.

Optimization stops automatically at the end of a block of assembly language statements.

NOTE: the compiler performs only peephole and scheduling optimizations on inline assembly language statements.

Instructs the compiler to stop optimizing inline assembly language instructions.

**Directive**  |  **Description**
---|---
optimize | Instructs the compiler to optimize the inline assembly language instructions that follow. You can turn off optimization within a block of assembly language statements using the nooptimize keyword. Optimization stops automatically at the end of a block of assembly language statements. NOTE: the compiler performs only peephole and scheduling optimizations on inline assembly language statements.
nooptimize | Instructs the compiler to stop optimizing inline assembly language instructions.
Comments

You can use single-line or multi-line comments with inline assembly language statements.

Single-line comments must start with a semicolon ( ; ) or a double slash ( // ). See Listing 6.12.

Listing 6.12  Single-line Comments in Inline Assembly Language

```assembly
asm {
    mov eax, 0 ; Single line comment - semicolon
    mov ebx, 1 // Single line comment - double slash
}
```

Multi-line comments must be bounded by slash-asterisk ( /* ), asterisk-slash ( */ ) pairs. See Listing 6.13.

Listing 6.13  Multi-line Comments in Inline Assembly Language

```assembly
asm {
    mov eax, 0
    /* the following instructions are commented out */
    mov ebx, 1
    mov ecx, 2
    */
}
```

NOTE  You cannot begin an inline assembly language comment with a pound sign (#). This is because all identifiers whose first character is a pound sign are handled by the preprocessor.

Preprocessor Support

The NetWare/x86 inline assembler supports all preprocessor features that the C/C++ compiler supports.

Refer to the CodeWarrior™ C Compilers Reference manual for preprocessor documentation.
Valid Instruction Operands

You can use the following as operands in inline assembly language statements:

- Local variable names
- Global variable names
- Structure, union, and class tags
  When you do this, the inline assembler uses the size of the structure, union, or class as the value of the operand.
- Structure, union, and class members.
  To do this, enter the name of the structure, union, or class, followed by a period (.), followed by the member name.

Listing 6.14 shows code that uses each type of operand.

Listing 6.14  Valid Instruction Operands

```c
struct MY_STRUCT {
    int member1;
    int member2;
} myStruct = { 3, 4 };

int myGlobal = 2;

void myFunc(void) {
    int myLocal = 1;

    asm {
        mov eax, myLocal  // eax contains 1
        mov ebx, myGlobal // ebx contains 2
        mov ecx, MY_STRUCT // ecx contains size of MY_STRUCT
        mov edx, myStruct.member2 // edx contains 4
        lea ebx, myStruct
        mov eax, MY_STRUCT.member1[ebx] // eax contains 3
    } // end assembly language block
} // end myFunc
```
Local Variables and MMX Registers

You can use local variables as operands for MMX™ and 3DNow!™ inline assembly language instructions.

The compiler can also assign local variables to MMX registers.

For a local variable to be a candidate for assignment to an MMX register, all of the following must be true:

- The function must not take the address of the local variable using the & operator.
- The function must not use the local variable in a non-MMX context.

A function uses a local variable in a non-MMX context if it uses the variable in any instruction that cannot accept an MMX register as an operand.

- For Intel processors, the local variable must be a 64-bit integer.
  Use the long long data type to declare a 64-bit integer.
- For AMD processors, the local variable must be a 64-bit integer or a float.

NOTE

The compiler can assign a float to a MMX register on an AMD processor only if you enable the 3DNow! extended instruction set in the x86 CodeGen target settings panel.

See x86 CodeGen Panel for an explanation of each item on the panel.

Listing 6.15 shows a function whose local variable is a candidate for assignment to an MMX register.
Listing 6.15  Local Variable Assigned to an MMX Register

```c
inline long long paddsb(long long a, long long b) {
    long long c; // c assigned to an MMX register
    asm {
        movq c, a
        paddsb c, b
    }
    return c;
}
```

Listing 6.16 shows a function whose local variable cannot be assigned by the compiler to an MMX register.

Listing 6.16 Assignment of a Local Variable to an MMX Register

```c
inline long long paddsb(long long mmx_reg_a, long long mmx_reg_b) {
    long long c;
    asm {
        mov eax, c // c cannot be assigned to an MMX reg because mov
        // does not allow an MMX register as its src operand
        movq c, a  // The movq instruction does allow an MMX register
        // as its source operand.
        paddsb c, b
    }
    return c;
}
```

Inlining Functions Containing `asm` Blocks

The NetWare/x86 C/C++ compiler can inline C and C++ functions that contain inline assembly language statements.

Optimizing Inline Assembly Language

The default behavior of the NetWare/x86 C/C++ compiler is not to optimize blocks of inline assembly language statements.
This is because the compiler assumes your assembly language code is already optimized. Otherwise, you would not have switched from C/C++ to assembly language in the first place.

However, you can instruct the compiler to optimize a block of assembly language statements by using the `optimize` directive.

When the compiler encounters the `optimize` directive, it optimizes all instructions up to the `nooptimize` directive or the closing brace of the assembly language block, whichever comes first.

**NOTE**

When you use the `optimize` directive, the compiler performs only scheduling and peephole optimizations. The compiler does not perform optimizations selected in the Global Optimization target settings panel.

Listing 6.17 shows a function that uses the `optimize` directive.

```
Listing 6.17  The optimize Directive

void myFunc(void) {
    int myLocal = 1;

    asm {
        optimize    //use of the optimize directive
        mov eax, myLocal
        mov ebx, myLocal
        mov ecx, myLocal
        mov edx, myLocal
    }            //optimization ends here
}               //end myFunc
```

**Working with x86 Assembly Language for AMD Processors**

All of the NetWare/x86 C/C++ compiler’s inline assembly language features work for both Intel and Advanced Micro Devices processors.
In addition, the inline assembler lets you use instructions that are specific to AMD processors, such as instructions that implement the 3DNow™ multimedia extensions.

**Targeting AMD Processors** documents features of the CodeWarrior Add-on Tools for NetWare that are specific to AMD processors.
80x86 Assembly Language Support
Working with x86 Assembly Language for AMD Processors

TNW–106 Targeting the NetWare® Operating System, PDK 5.0
The NetWare/x86 Linker

This chapter explains the features of the NetWare®/x86 linker.

This chapter contains these sections:

- Linker Input Files
- Linker Output Files
- Dynamic Linking
- Static Linking
- Deadstripping
- Link Order

Linker Input Files

The NetWare/x86 linker can process these types of input files:

- Object Modules
- Static Libraries
- NLM Files
- Linker Command Files
- Symbol Export Files
- Symbol Import Files
- Link Order Files
- Help Screen Files
- Message Files
- Remote Procedure Call Files
Object Modules

The NetWare/x86 linker can link object modules into your NLM.

An object module is a file created by the compiler using the source code in a single source file.

The linker can read these object module types:

- COFF (Common Object File Format) object modules produced by the NetWare/x86 C/C++ compiler.
- COFF object modules that you add to your project.

Static Libraries

The NetWare/x86 linker can read static COFF static library files that you have added to your project.

A static library is a collection of object modules that the linker can copy into your NLM.

NLM Files

You can add NLM files to your project.

The linker uses these NLMs to satisfy unresolved external references made by your project’s NLM.

NOTE

If an NLM has an import file, add the import file to your project instead of the NLM itself.

This is because the linker cannot read symbols from a packed or compressed NLM. However, the linker can resolve references to symbols in a packed or compressed NLM using that NLM’s import file.
Linker Command Files

The NetWare/x86 linker can read and act upon a linker command file.

A linker command file is a text file. It contains commands that control characteristics of the output file the linker creates.

For the linker to use a linker command file, you must add the file to your project. The name of a linker command file must always end with the suffix .DEF.

---

NOTE

A project can contain any number of linker command files or no command file.

The linker processes each command file in the order it appears in the Link Order view of your project. A command that appears in a later file overrides the same command found in an earlier file.

---

[NetWare Linker Command Files](#) lists and defines commands allowed in a linker command file.

Symbol Export Files

The NetWare/x86 linker can read and act upon a symbol export file.

A symbol export file is a text file. It lists the names of variables and functions in your NLM that you want to make available to other NLMs.

For the linker to use a symbol export file, you must add the file to your project. The name of a symbol export file must always end with the suffix .EXP.

A project can have more than one symbol export file or none at all.

---

NOTE

You can also export symbols using the **EXPORT** command in a linker command file.
Symbol Prefixes in Export Files

To allow developers to use libraries from different vendors, Novell’s export file syntax allows symbols to include a prefix.

The prefix uniquely identifies the publisher of the library. For example, Novell uses the prefix NOVL in its libraries.

There are two ways to prefix exported symbols:

- Precede a group of symbols with the prefix surrounded by parentheses. This technique is shown in Listing 7.1
- Precede an individual symbol with the prefix followed by the at character (@). This technique is shown in Listing 7.2.

Listing 7.1 Using an Export Symbol Prefix for a Group of Symbols

```
EXPORT symbol1
(NOVL)
symbol2
symbol3
```

Listing 7.2 Using an Export Symbol Prefix for a Single Symbol

```
EXPORT symbol1
NOVL@symbol2
symbol3
```

For a more detailed discussion of the purpose of prefixes and how they work, see the NetWare documentation.

Symbol Import Files

The NetWare/x86 linker can read and act upon a symbol import file.

A symbol import file is a text file. It lists the names of variables and functions defined in other NLMs that your NLM uses.

For the linker to use a symbol import file, you must add the file to your project.
The name of a symbol import file must always end with the suffix .IMP.

Add an import file to your project for each NLM that defines variables or functions that your NLM uses. NetWare automatically loads any NLM referenced in an import file when it loads your NLM.

**NOTE**
You can also import symbols using the `IMPORT` command in a linker command file.

### Symbol Prefixes in Symbol Import Files

To allow developers to use libraries from different vendors, Novell’s import file syntax allows symbols to include a prefix.

The prefix uniquely identifies the publisher of the library. For example, Novell uses the prefix `NOVL` in its libraries.

There are two ways to prefix imported symbols:

- Precede a group of symbols with the prefix surrounded by parentheses. This technique is shown in **Listing 7.3**
- Precede an individual symbol with the prefix followed by the at character (@). This technique is shown in **Listing 7.4**.

**Listing 7.3** Using an Import Symbol Prefix for a Group of Symbols

```
IMPORT symbol1
(NOVL)
symbol2
symbol3
```

**Listing 7.4** Using an Import Symbol Prefix for a Single Symbol

```
IMPORT symbol1
NOVL@symbol2
symbol3
```

For a more detailed discussion of the purpose of prefixes and how they work, see the NetWare documentation.
Link Order Files

You can add files ending with the suffix .ORD to your project. These files contain names of symbols the linker should link first.

NOTE
You can use the LINKORDER command in a linker command file instead of adding a .ORD file to your project.

Help Screen Files

You can add one file ending with the suffix .HLP to your project. This file contains the help screens displayed by your NLM.

NOTE
You can use the HELP command in a linker command file instead of adding a .HLP file to your project.

Message Files

One file ending with the suffix .MSG can be added to your project. This file contains messages displayed by your NLM.

NOTE
You can use the MESSAGES command in a linker command file instead of adding a .MSG file to your project.

Remote Procedure Call Files

One file with a .RPC extension can be added to your project. This file contains remote procedure call descriptions that allow functions to be exported across memory protection boundaries. The linker puts the path and name of a project’s .RPC file in the NLM header.

NOTE
You can use the XDCDATA command in a linker command file instead of adding a .RPC file to your project.
The RPC compiler (which produces RPC files) has not yet been released.
Linker Output Files

The linker can generate these types of output files:

- **NetWare Loadable Modules**
- **NLM Map Files**

**NetWare Loadable Modules**

The most important file created by the NetWare/x86 linker is called a NetWare Loadable Module™ (NLM). An NLM is a file that the NetWare operating system can load and execute.

The NetWare/x86 linker can create these types of NLMs:

- Generic NLMs (.NLM files)
- LAN Drivers (.LAN files)
- Disk Drivers (.DSK files)
- Name Space Support (.NAM files)
- Utilities (.NLM files)
- Mirrored Server Link (.MSL files)
- Operating System (.NLM files)
- Paged High Operating System (.NLM files)
- Host Adapters (.HAM files)
- Custom Device (.CDM files)
- Object Library (.LIB files)

**NLM Map Files**

The NetWare/x86 linker can create a map file for your NLM. A map file shows the memory address, size, and module of declaration for every variable and function in your NLM. Using the information in a map file, you can often identify the routine that was executing when your NLM crashed.

To create a map file, enable the Generate Link Map item in the NLM Linker target settings panel. For more information on this item, see “NLM Linker Panel” on page 57.
The NetWare/x86 Linker
Linker Output Files

The linker places the map file in your project directory. All map file names end with .MAP. Listing 7.5 shows an example of a map file, which has been modified. Examine a map file in your project directory for a true representation.

### Listing 7.5  Example of a Map File

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Name</th>
<th>Subname</th>
<th>Flags</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00000015</td>
<td>NLM_Code</td>
<td></td>
<td></td>
<td>HelloWorld.c(main)</td>
</tr>
<tr>
<td>00000015</td>
<td>00000012</td>
<td>NLM_Code</td>
<td>clibpre.o(<em>cstart</em>)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000027</td>
<td>00000077</td>
<td>NLM_Code</td>
<td>clibpre.o(_Prelude)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000009e</td>
<td>0000001f</td>
<td>NLM_Code</td>
<td>clibpre.o(_Stop)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000c0</td>
<td>00000020</td>
<td>NLM_Code</td>
<td>MWStart.obj(_init_environment)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000e0</td>
<td>0000000f</td>
<td>NLM_Code</td>
<td>MWStart.obj(_deinit_environment)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000000f0</td>
<td>00000075</td>
<td>NLM_Code</td>
<td>ExceptionX86.obj(_RegisterExceptionTables)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000006d</td>
<td>00000000</td>
<td>NLM_Code</td>
<td>D ExceptionX86.obj(_UnRegisterExceptionTables)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>NLM_Code</td>
<td>D ExceptionX86.obj(_init__catch)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>00000000</td>
<td>NLM_Code</td>
<td>D ExceptionX86.obj(_end__catch)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000170</td>
<td>0000000f</td>
<td>NLM_Code</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000180</td>
<td>0000003d</td>
<td>NLM_Code</td>
<td>runinit.obj(_RunInit)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001c0</td>
<td>0000004b</td>
<td>NLM_Code</td>
<td>D globdest.obj(_detect_cpu_instruction_set)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000210</td>
<td>00000034</td>
<td>NLM_Code</td>
<td>globdest.obj(_register_global_object)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000021d</td>
<td>0000004d</td>
<td>NLM_Code</td>
<td>D globdest.obj(_destroy_global_chain)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000250</td>
<td>0000000a</td>
<td>NLM_Code</td>
<td>S NMWException.obj(?dthandler%std%%YAXXZ 'void std::dthandler(void)')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000260</td>
<td>0000000a</td>
<td>NLM_Code</td>
<td>S NMWException.obj(?duhandler%std%%YAXXZ 'void std::duhandler(void)')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000270</td>
<td>0000000b</td>
<td>NLM_Code</td>
<td>NMWException.obj(terminate%std%%YAXXZ 'void std::terminate(void)')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000320</td>
<td>00000000</td>
<td>NLM_Code</td>
<td>D destroy.obj(_destroy_new_array)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000320</td>
<td>00000000</td>
<td>NLM_Code</td>
<td>D destroy.obj(_destroy_new_array2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000320</td>
<td>00000000</td>
<td>NLM_Code</td>
<td>D destroy.obj(_destroy_new_array3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000320</td>
<td>00000000</td>
<td>NLM_Code</td>
<td>D delop.obj(??3%YAXPAX%Z 'void operator delete(void *)')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000320</td>
<td>00000000</td>
<td>NLM_Code</td>
<td>D delarrop.obj(??_V%YAXPAX%Z 'void operator delete[](void *)')</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000380</td>
<td>00000020</td>
<td>NLM_Data</td>
<td>HelloWorld.c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000380</td>
<td>00000018</td>
<td>NLM_Data</td>
<td>clibpre.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000480</td>
<td>00000178</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000004c8</td>
<td>00000008</td>
<td>NLM_Data</td>
<td>NMWException.obj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005d0</td>
<td>00000001</td>
<td>NLM_Data</td>
<td>clibpre.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005d4</td>
<td>00000001</td>
<td>NLM_Data</td>
<td>clibpre.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005d8</td>
<td>00000001</td>
<td>NLM_Data</td>
<td>clibpre.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005d9</td>
<td>00000001</td>
<td>NLM_Data</td>
<td>MWStart.obj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005da</td>
<td>00000001</td>
<td>NLM_Data</td>
<td>MWStart.obj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005db</td>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005de</td>
<td>00000001</td>
<td>NLM_Data</td>
<td>D ExceptionX86.obj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005f8</td>
<td>00000008</td>
<td>NLM_Data</td>
<td>SD ExceptionX86.obj(0176)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000005f8</td>
<td>00000001</td>
<td>NLM_Data</td>
<td>D ExceptionX86.obj</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Address</td>
<td>Type</td>
<td>File Name</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>--------</td>
<td>--------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000004c</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj (0218)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001c</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj (0265)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001c</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj (0290)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000004c</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj (0435)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000091</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000043</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000015</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000019</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000050</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj (0619)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000050</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj (0618)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000050</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj (0682)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000015</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001dc</td>
<td>NLM_Data</td>
<td>ExceptionX86.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001dd</td>
<td>NLM_Data</td>
<td>runinit.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000001de</td>
<td>NLM_Data</td>
<td>runinit.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>globdest.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>globdest.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001df</td>
<td>NLM_Data</td>
<td>globdest.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>globdest.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001e0</td>
<td>NLM_Data</td>
<td>NMWException.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001e1</td>
<td>NLM_Data</td>
<td>NMWException.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>NMWException.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000001e2</td>
<td>NLM_Data</td>
<td>NMWException.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>NMWException.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000028</td>
<td>NLM_Data</td>
<td>NMWException.obj (0167)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>NMWException.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>destroy.obj</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### The NetWare/x86 Linker
#### Linker Output Files

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Type</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>D</td>
<td>destroy.obj</td>
</tr>
<tr>
<td>00000029</td>
<td>NLM_Data</td>
<td>D</td>
<td>delop.obj</td>
</tr>
<tr>
<td>00000001</td>
<td>NLM_Data</td>
<td>D</td>
<td>delarrop.obj</td>
</tr>
<tr>
<td>00000143</td>
<td>NLM_Data</td>
<td>$AAA</td>
<td>cexc.obj</td>
</tr>
<tr>
<td>00000008</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>clibpre.o</td>
</tr>
<tr>
<td>00000144</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>clibpre.o</td>
</tr>
<tr>
<td>000001f0</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>MWStart.obj</td>
</tr>
<tr>
<td>00000204</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>MWStart.obj</td>
</tr>
<tr>
<td>0000020c</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>ExceptionX86.obj</td>
</tr>
<tr>
<td>00000214</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>ExceptionX86.obj</td>
</tr>
<tr>
<td>00000224</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>globdest.obj</td>
</tr>
<tr>
<td>0000022c</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>globdest.obj</td>
</tr>
<tr>
<td>00000234</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>NMWException.obj</td>
</tr>
<tr>
<td>0000023c</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>NMWException.obj</td>
</tr>
<tr>
<td>00000240</td>
<td>NLM_Data</td>
<td>$BBB</td>
<td>globdest.obj</td>
</tr>
</tbody>
</table>

---

**TNW–116** Targeting the NetWare® Operating System, PDK 5.0
The NetWare/x86 Linker
Linker Output Files

Targeting the NetWare® Operating System, PDK 5.0

00000244 00000008 NLM_Data $BBB NMWException.obj
00000008 NLM_Data $BBB D NMWException.obj
00000008 NLM_Data $BBB D NMWException.obj
00000008 NLM_Data $BBB D NMWException.obj
00000008 NLM_Data $BBB D destroy.obj
00000008 NLM_Data $BBB D destroy.obj
00000008 NLM_Data $BBB D destroy.obj
00000008 NLM_Data $BBB D delop.obj
00000008 NLM_Data $BBB D delarrop.obj
0000024c 00000004 NLM_Data $CCC cexc.obj
00000250 00000004 NLM_Data $XCA ccinit.obj
00000254 00000004 NLM_Data $XCU ExceptionX86.obj
00000258 00000004 NLM_Data $XCZ ccinit.obj
00000260 00000004 NLM_BSS ExceptionX86.obj
00000268 00000304 NLM_BSS globdest.obj
00000570 00000004 NLM_BSS

Legend:
--------
S: static code
D: dead code

----------------
Public Symbols
----------------

<table>
<thead>
<tr>
<th>Type</th>
<th>Address</th>
<th>Module</th>
<th>Flags</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>00000000</td>
<td>HelloWorld.c</td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>CODE</td>
<td>00000015</td>
<td>clibpre.o</td>
<td></td>
<td><em>cstart</em></td>
</tr>
<tr>
<td>CODE</td>
<td>00000027</td>
<td>clibpre.o</td>
<td></td>
<td>_Prelude</td>
</tr>
<tr>
<td>CODE</td>
<td>0000009e</td>
<td>clibpre.o</td>
<td></td>
<td>_Stop</td>
</tr>
<tr>
<td>CODE</td>
<td>000000f0</td>
<td>ExceptionX86.obj</td>
<td></td>
<td>_RegisterExceptionTables</td>
</tr>
<tr>
<td>CODE</td>
<td>00000180</td>
<td>runinit.obj</td>
<td></td>
<td>_RunInit</td>
</tr>
<tr>
<td>CODE</td>
<td>000001c0</td>
<td>runinit.obj</td>
<td></td>
<td>__detect_cpu_instruction_set</td>
</tr>
<tr>
<td>CODE</td>
<td>00000210</td>
<td>globdest.obj</td>
<td></td>
<td>__destroy_global_chain</td>
</tr>
</tbody>
</table>
| CODE | 00000270 | NMWException.obj     |       | ?terminate%std%%YAXXZ 'void std::terminate(void)'
| DATA | 00000020 | clibpre.o            |       | __argc                |
| DATA | 00000024 | clibpre.o            |       | kNLMInfo              |
| DATA | 000001c0 | ExceptionX86.obj     |       | ??_R0?AVexception%std%%%8~ 'class std::exception`MW RTTI Type Descriptor'' |
| DATA | 000001e4 | cexc.obj             |       | __PcToActionStart     |
| DATA | 0000024c | cexc.obj             |       | __PcToActionEnd       |
| DATA | 00000250 | ccinit.obj           |       | __xc_a                |
| DATA | 00000258 | ccinit.obj           |       | __xc_z                |
| DATA | 0000025c | common               |       | __NLM_BSS_Start       |
| DATA | 00000570 | common               |       | __global_destructor_chain |
| DATA | 00000574 | common               |       | __NLM_BSS_End         |

-----------------
Import/Export Symbols
-----------------
The NetWare/x86 Linker

Linker Output Files

<table>
<thead>
<tr>
<th>Import/Export</th>
<th>Symbol Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Import]</td>
<td>[clib@]memset</td>
</tr>
<tr>
<td>[Import]</td>
<td>[clib@]._SetupArgV_411</td>
</tr>
<tr>
<td>[Import]</td>
<td>[clib@]printf</td>
</tr>
<tr>
<td>[Import]</td>
<td>[clib@]abort</td>
</tr>
<tr>
<td>[Import]</td>
<td>[clib@]._StartNLM</td>
</tr>
<tr>
<td>[Import]</td>
<td>[clib@]._TerminateNLM</td>
</tr>
<tr>
<td>[Import]</td>
<td>[clib@]__init_environment</td>
</tr>
<tr>
<td>[Import]</td>
<td>[clib@]__deinit_environment</td>
</tr>
</tbody>
</table>

Compiler/Linker Switches

C/C++ Language

- Force C++ compilation: OFF
- Require prototypes: OFF
- ISO Template Parser: OFF
- Enable C99 Extensions: OFF
- Enable Objective C: OFF
- Legacy for-scoping: OFF
- Enable C++ Exceptions: OFF
- Enable RTTI: OFF
- Enable bool Support: ON
- Enable wchar_type Support: ON
- Inline level: Smart
- Auto-Inline: OFF
- Deferred Inlining: OFF
- Bottom-up Inlining: ON
- ANSI Strict: OFF
- ANSI KeyWords Only: OFF
- Expand Trigraphs: OFF
- Enable C99 Extensions: OFF
- Multi-Byte Aware: OFF
- EC++ Compatibility Mode: OFF
- Map newlines to CR: OFF
- Enums Always Int: OFF
- Use Unsigned Char: OFF
- Pool Strings: OFF
- Reuse Strings: ON
- Require Function Prototypes: OFF
- Prefix File: nlm_prefix.h

Global Optimizations

- Optimize For: Faster Execution Speed
- Optimize For: Level 0

x86 CodeGen

- Runtime configuration: Custom
Target processor: Generic
Processor-specific instructions: OFF
MMX: OFF
3DNow!: OFF
Conditional moves: OFF
SSE: OFF
SSE-2: OFF
Use multimedia calling convention: OFF
Make strings read-only: OFF
Struct alignment: 1
Inline intrinsic functions: OFF
Optimize inline assembly: OFF
Disable optimizations for debug: OFF
Microsoft-compatible name mangling: OFF
Microsoft-compatible C++ exception handling: OFF

NLM Linker Panel
---------------------
Check:
Start: _Prelude
Exit : _Stop
Generate Link Map: ON
Generate Codeview Debug Info: ON
Generate Internal Debugger Record: ON
Expand Uninitialized Data: ON
Flags: (none)
Suppress Warning Messages: OFF
Verbose Messages: OFF
Store full paths: OFF

NLM Target
------------
Output Type: Generic NLM (.NLM)
Output File: C_CLibd.nlm
Copyright: (C) Copyright Your Name Here
Description: Generic NLM Stationery
Initial Screen: DEFAULT
Initial Thread:
Stack: 8192
Major: 1
Minor: 0
Revision: 0

NLM filename: C_CLibd.nlm
Version: 1.0    2002-AUG-27
Description: Generic NLM Stationery
Copyright: (C) Copyright Your Name Here
NLM last modify date/time: Tue Aug 27 17:53:37 2002
MD4 Checksum: 4ad4a2eb07e5ea09953fed12b90b2044
Dynamic Linking

A shared library NLM is a collection of functions and data that your NLM can use.

Each time you run your NLM, NetWare loads each shared library NLM that contains functions your NLM uses. Your NLM then links to the functions it uses at runtime. This is called **dynamic linking**.

To use a shared library NLM, follow these steps:

1. Include a header file that prototypes the shared library’s functions in each of your NLM’s source code files that use the library.
2. Use the shared library functions and data you need in the NLM source code files in which you included the library header file.
3. Create an import file for the shared library.
   The name of this file must end with the suffix `.IMP`. The root part of the file name can be anything.
4. Enter the names of the shared library functions and data you used in your NLM in the import file.
5. Add the import file to your project.
6. Build your project.
   The linker adds references to the shared library NLM and the functions that your NLM uses to your NLM’s header.

Your projects can use shared library NLMs included with the CodeWarrior Add-on Tools for NetWare, third party shared library NLMs, and shared library NLMs you create yourself.

Static Linking

A **static library** is a collection of functions and data that your NLM can use.

Each time you build your NLM, the linker copies the functions and data your NLM uses from the static library into your NLM. This is called **static linking**.

To use a static library, follow these steps:
1. Add the static library to your project.
2. Include the header file that prototypes the library’s functions in each of your NLM’s source code files that use the library.
3. Reference the library functions and data you need in the NLM source code files in which you included the library header file.
4. Build your project.
   The linker copies the code and data your NLM needs from the static library into your NLM.

Your projects can use static libraries included with the CodeWarrior Add-on Tools for NetWare, third party static libraries, and static libraries you create yourself.

**Deadstripping**

As it builds its output file, the NetWare/x86 linker removes all code and data from the project’s object files that your NLM does not use. This is called *deadstripping*.

The linker can deadstrip only object files created by the CodeWarrior NetWare/x86 C/C++ compiler.

The linker does not deadstrip C/C++ object files created by third party compilers.

**Link Order**

In most cases, you define a project’s link order in the Link Order view of the project window.

Figure 7.1 shows the Link Order view of a project window.
Regardless of the link order you define in the Link Order view, the NetWare/x86 linker resolves external references by searching files in this order:

1. Object modules generated by the compiler.
2. Static libraries.
3. Files that list imported symbols. These files are:
   - Import files added to the project.
   - NLMs added to the project.
   - Linker command files that contain IMPORT commands added to the project.

This means that if the same symbol is defined in a source file and a library, the linker uses the definition in the source file instead of the definition in the library.
Command Line Tools

This chapter explains how to use the command line compiler and command line linker. In particular, the chapter lists and defines the options these tools accept.

Some command line options contain an equivalent setting in a target setting panel. These options and their equivalent panel setting are documented.

From a DOS prompt, you can enter `mwccnlm -help` to view information about compiler command line options or `mwldnlm -help` for linker command line options.

Contained in this chapter are

- Conventions
- Global Command Line Options
- Compiler Options
- Linker Options

Conventions

When an option is specified as `-xxx | yy[y] | zzz`, then `-xxx`, `-yy`, `-yyy`, or `-zzz` matches the option.

An option given as `- [no]xxx` may be given as `-xxx` or `-noxxx`; `-noxxx` reverses the meaning of the option.

For most options, the option and parameters are separated by a space.

When the option’s name is `-xxx+`, however, the parameter must directly follow the option, without the plus `+` (for example, `-xxx45`).

A parameter included in brackets `([ ])` is optional. An ellipsis `(...)` indicates that the previous type of parameter may be repeated as a list.
Conventions

Table 8.1 Command Line Options Term Definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cased</td>
<td>Indicates that the option is case-sensitive. By default, no options are case-sensitive.</td>
</tr>
<tr>
<td>Compatibility</td>
<td>Indicates that the option is borrowed from another vendor’s tool and may only approximate its counterpart.</td>
</tr>
<tr>
<td>Global</td>
<td>Indicates that the option has an effect over the entire command line and is parsed before any other options. When several global options are specified, they are interpreted in order from right to left.</td>
</tr>
<tr>
<td>Deprecated</td>
<td>Indicates that the option will be eliminated in the future and should not be used any longer. A new, alternative options exists.</td>
</tr>
<tr>
<td>Ignored</td>
<td>Indicates that the option is accepted but has no affect on the tool.</td>
</tr>
<tr>
<td>Meaningless</td>
<td>Indicates the option is accepted but probably has no meaning for the target OS.</td>
</tr>
<tr>
<td>Obsolete</td>
<td>Indicates the option was once deprecated and is now gone.</td>
</tr>
<tr>
<td>Substituted</td>
<td>Indicates the option has the same effect as another. This points out a preferred form and prevents confusion when similar options appear in the help.</td>
</tr>
<tr>
<td>Default</td>
<td>Indicates that the given value or variation of an option is used unless overridden.</td>
</tr>
</tbody>
</table>

The comma (,) and equal (=) characters are used to separate options and parameters unconditionally. To include one of these symbols in a parameter or filename, prefix it with the escape character (\).

This command line compiler calls the command line linker (unless a compiler option (such as -c) prevents it. The command line compiler passes linker options to the command line linker.

Options marked “passed to linker” are used by the compiler and linker; options marked “for linker” are used only by the linker. When using the compiler and linker separately, you must pass the common options to both.
Global Command Line Options

All options defined in this section are passed to the linker unless otherwise noted.

-help [keyword[,...]]

Global. Displays help for the command line compiler and command line linker.

The options for keyword are:

- usage
  Show explanation of conventions used in the help information.
- [no]spaces
  Insert blank lines between options in a printout.
- all
  Show all standard options.
- [no]normal
  Show only standard options.
- [no]obsolete
  Show obsolete options.
- [no]ignored
  Show ignored options.
- [no]deprecated
  Show deprecated options.
- [no]meaningless
  Show options that are meaningless for this target.
- [no]compatible
  Show compatibility options.
- opt[ion]=name
  Show help for a given option. The maximum number of characters for name is 63 characters.
- search=keyword
  Show help for an option whose name or help contains keyword (case-sensitive). The maximum number of characters for keyword is 63.
• **group=keyword**
  Show help for groups whose names contain *keyword* (case-sensitive). The maximum number of characters for *keyword* is 63.

• **tool=keyword[,...]**
  Categorize groups of options by tool. Default.
  The options for *keyword* are:
  - **all**
    Show all options available for this tool.
  - **this**
    Show options acted upon by this tool. Default.
  - **other | skipped**
    Show options that are passed through to another tool.
  - **both**
    Show options used by both the compiler and linker.

**-version**
Global. Display the version number of the compiler or linker. The version number consists of: version, configuration, and build date.

**-timing**

**-progress**
Global. Show progress and version.

**-v[erbose]**

This option echoes messages that would appear in the IDE’s Build Progress window, but the command line tool also emits its own messages.
**-search**

Global. Search access paths for source files specified on the command line. You can specify object modules and libraries, as well. This option provides the access paths functionality of the CodeWarrior IDE.

This option is similar to the “Save project entries using relative paths” checkbox in the Target Settings panel. The default behavior assumes that a file on the command line is located relative to the current directory. An error is issued if the file cannot be found. This option also enables searching in paths assigned to the -I or -L flags, if it cannot be found in the current directory.

**-[no]wraplines**


**-maxerrors max**

Specify maximum number of errors to print. Zero (0) means no maximum. The default is 0.

**-maxwarnings max**

Specify maximum number of warnings to print. Zero (0) means no maximum. The default is 0.

**-msgstyle keyword**

Global. Set the message style for errors and warnings.

The options for keyword are:

- **mpw** - Use MPW message style
- **std** - Use standard message style (Default)
- **gcc** - Use GCC message style.
- **IDE** - Use CodeWarrior IDE message style.
- **parseable** - Use context free, machine-parseable message style.

**-[no]stderr**

Global. Use separate stderr and stdout streams. If you use -nostderr, stderr goes to stdout.
Compiler Options

- Preprocessing, Precompiling, and Input File Control Options
- C/C++ Language Options
- x86 Code Generation Options
- Code Optimization Options
- C/C++ Warning Options
- Debugging Control Options

Preprocessing, Precompiling, and Input File Control Options

-c
Global. Compile only, do not link.

IDE Equivalent: Project > Compile

-[no]codegen
Global. Generate object code.

IDE Equivalent: When enabled, Project > Compile, otherwise Project > Check Syntax

-[no]convertpaths
Global. Default. Interpret #include file paths specified for a foreign operating system. That is, <sys/stat.h> or <:sys:stat.h>. When enabled, (/) and (:) separate directories and cannot be used in filenames.

NOTE: This is not a problem on Windows since these characters are already disallowed in file names. It is safe to leave this option on.

IDE Equivalent: Edit > targetname Settings > Access Paths > Interpret DOS/UNIX Paths (this option only exists in the MacOS or Unix version of the IDE)

-cwd keyword
Specify #include searching semantics. The path represented by the keyword is searched before searching access paths defined for the build target.
The options for *keyword* are:

- **proj**
  Default. Begin search in current working directory.

- **source**
  Begin search in directory that contains the source file.

- **explicit**
  No implicit directory. Search `-I` or `-ir` paths.

- **include**
  Begin search in directory of referencing file.

**-D+ | -d[efine]name[=value]**

Cased. If specified, define *name* to *value*. Otherwise, define *name* to 1.

IDE Equivalent: Edit > targetname Settings > C/C++ Language > Prefix file (Use #define statement in prefix file.)

**-[no]defaults**


The command line tool uses additional environment variables to provide defaults. In the IDE, all access paths and libraries are explicit.

**-dis[assemble]**

Global. Passed to all tools. Disassemble files and send result to `stdout`.

IDE Equivalent: Project > Disassemble (the IDE always sends output to a new unsaved text editor window)

**-E**


IDE Equivalent: Project > Preprocess (the IDE sends output to a new unsaved text editor window)

**-EP**

Global. Cased. Preprocess and strip out `#line` directives.

IDE Equivalent: Project > Preprocess, with a "#pragma simple_prepdump on". The IDE sends files to a new unsaved editor window.
**-ext extension**

Global. Specify the extension for generated object files. If the value specified for `extension` includes a leading period (.), the extension is appended to the name of each object file generated. If the value specified for `extension` does not start with a period, the specified extension replaces the source file’s extension. The maximum length for `extension` is 14 characters. The default is none.

The IDE writes object files to the target data directory and is not configurable.

**-gccinc[ludes]**

Global. Adopt GCC `#include` semantics. The GCC include semantics are: add `-I` paths to system list if `-I-` is not specified and search directory of the referencing file first for included files (same as `-cwd include`).

The compiler and IDE only search access paths, and do not take the currently #included file into account.

**-i- | -I-**

Global. Change target for `-I` access paths to the system list. Implies `-cwd explicit`. For include statements of the form `#include "xyz"`, the compiler first searches user paths and then system paths. For include statements of the form `#include <xyz>`, the compiler searches only system paths.

**IDE Equivalent**

Add access paths as needed. Choose Edit > `targetname` Settings... > Access paths and then add User paths or System Paths.

**-I+ | -i path**

Global. Cased. Append access path to current `#include` list. See `-gccinclusions` and `-I-`.

**IDE Equivalent**

Add non-recursive access paths. Choose Edit > `targetname` Settings... > Access paths.

**-include file**

Add text file or precompiled header as a prefix to all source files.

**IDE Equivalent**

Choose Edit > `targetname` Settings... > C/C++ Language. In the Prefix file text box enter the name of the file you create that `#includes` other prefix files. With the command line tool, you can add multiple prefix files (all of which are included in a meta-prefix file).
-ir path

Global. Append a recursive access path to current #include list.

**IDE Equivalent**

Choose Edit > targetname Settings... > Access Paths and add recursive access paths.

-[no]keepobjects

Global. Keep object files generated after invoking linker. If disabled, intermediate object files are deleted after the link stage. Objects are always kept when compiling.

-M

Global. Cased. Scan source files for dependencies and emit a Makefile. Do not generate object code.

-MM

Global. Cased. Like -M, but does not list system #include files.

-MD

Global. Cased. Like -M, but writes dependency map to a file and generates object code.

-MMD

Global. Cased. Like -MD, but does not list system #include files.

-make

Global. Scan source files for dependencies and write a Makefile. Do not generate object code. The IDE keeps track of dependencies internally and are not readily available.

-nofail

Continue working in spite of errors in previous files.

-nolink

Global. Compile only; do not link.

**IDE Equivalent**

Select all source files and Choose Project > Compile.
-noprecompile
Do not precompile any files based on the filename extension.

IDE Equivalent
Choose Edit > targetname Settings... > File Mappings and uncheck the "Precompiled" flag (in Flags dropdown list) for the selected extension.

-nosyspath
Global. Treat include statements of the form #include <xyz> the same as include statements of the form #include "xyz". In other words, always search both user and system path lists.

IDE Equivalent
Choose Edit > targetname Settings... > Access Paths and enable "Always search user paths."

-o file | dir
Specify output file name or directory for object files, text files, and the linker’s output file (if the linker is called after the compiler finishes). The IDE always stores object files at Project_Path\Project_Data\Target_Data\ObjectCode folder, with a name that cannot be configured.

-P

IDE Equivalent
Choose Project > Preprocess to send output to editor window. Then save file to disk.

-precompile file | dir
Generate precompiled header from source. Write header to file, if specified. Place header in dir, if specified.

If argument is " ", write header to location specified in source code. If neither argument is defined, the header file name is derived from the source file name.

NOTE
The driver can determine whether to precompile a file based on its extension. The statement -precompile filesourc e is equivalent to -c -o filesourc e.

IDE Equivalent
Choose Project > Precompile and save file to directory.
**-preprocess**

Global. Preprocess source files.

**IDE Equivalent**

Choose Project > Preprocess

**-prefix file**

Add contents of file (a text file or precompiled header) as a prefix to all source files.

**IDE Equivalent**

Choose Edit > targetname Settings... > C/C++ Language and specify text file or precompiled header in Prefix File text box.

**-S**

Global. Cased. Passed to all tools. Disassemble and send output to a file.

**IDE Equivalent**

Choose Project > Disassemble and save output to file.

**-[no]stdinc**

Global. Use standard system include paths (specified by the environment variable %MWCIncludes%. Add this option after all system -I paths. Default. The IDE does not use default paths or files; they are explicit in the project settings.

**-U+ | -u[ndefine] name**

Cased. Undefine the symbol name.

**IDE Equivalent**

Use #undef statement in prefix file specified in C/C++ Language target setting panel.

### C/C++ Language Options

**-ansi keyword**

Specify ANSI conformance options, overriding the given settings.

The options for keyword are:

- off
  
  Same as -stdkeywords off, -enum min, and -strict off

- on | relaxed
Same as -stdkeywords on, -enum min, and -strict on

- **strict**
  Same as -stdkeywords on, -enum int, and -strict on

**IDE Equivalent**
Enable “ANSI Strict” checkbox in C/C++ Language target setting panel. Choose Edit > targetname Settings... > C/C++ Language.

**-ARM on | off**
Check code for ARM (Annotated C++ Reference Manual) conformance. Default is off.

**IDE Equivalent**
Enable “Legacy for-scoping” checkbox in C/C++ Language target setting panel. Choose Edit > targetname Settings... > C/C++ Language.

**-bool on | off**
Allow true and false constants for the C++ bool data type. Default is off.

**IDE Equivalent**
Select “Enable bool support” in C/C++ Language target setting panel.

**-char keyword**
Set the default sign of the char data type.

The options for keyword are:

- **signed**
  char data items are signed. Default.
- **unsigned**
  char data items are unsigned.

**IDE Equivalent**
Select “Use Unsigned Chars” in C/C++ Language target setting panel.

**-Cpp_exceptions on | off**
Enable or disable C++ exceptions. Default is on.

**IDE Equivalent**
Select “Enable C++ Exceptions” in C/C++ Language target setting panel.

**-dialect | -lang keyword**
Specify source language.
The options for \textit{keyword} are:

- \texttt{c}
  
  Always treat source as C.

- \texttt{c++}
  
  Always treat source as C++.

**IDE Equivalent**

Select “Force C++ compilation” in C/C++ Language target setting panel.

- \texttt{ec++}
  
  Generate warnings for use of C++ features outside the Embedded C++ subset. Implies \texttt{dialect cplus}.

**IDE Equivalent**

Select “EC++ Compatibility Mode” in C/C++ Language target setting panel.

\textbf{-enum} \textit{keyword}

Specify the word size for enumerated types.

The options for \textit{keyword} are:

- \texttt{min}
  
  Use minimum sized enums. Default.

- \texttt{int}
  
  Use int-sized enums.

**IDE Equivalent**

Select “Enums Always Int” in C/C++ Language target setting panel.

\textbf{-inline} \textit{keyword}[,...]

Specify inline options.

The options for \textit{keyword} are:

- \texttt{on | smart}
  
  Turn on inlining for \texttt{inline} functions. Default.

**IDE Equivalent**

Select “Inline Depth: Smart” in the C/C++ Language target setting panel.

- \texttt{none | off}
  
  Turn off inlining.

**IDE Equivalent**

Select “Inline Depth: Don’t Inline” in C/C++ Language target setting panel.
Command Line Tools

Compiler Options

- noauto
  Do not auto-inline. Default.

- auto
  Auto-inline small functions (without inline explicitly specified).

  IDE Equivalent: Select “Auto-Inline” in C/C++ Language target setting panel.

- all
  Turn on aggressive inlining. This option is the same as -inline on, auto

- deferred
  Defer inlining until end of compilation unit. This allows inlining of functions in both directions.

  IDE Equivalent: Select “Deferred Inlining” in C/C++ Language target setting panel.

- level=n
  Cased. Inline functions up to n levels deep. Level 0 is the same as -inline on. For n, enter 1 to 8 levels.

  IDE Equivalent: Select “Inline Depth: (choose 1 to 8)” in C/C++ Language target setting panel.

-iso_templates on | off

  Enable ISO C++ template parser. Default is off.

  NOTE: This requires a different MSL C++ library.

  IDE Equivalent: Select “ISO C++ Template Parser” in C/C++ Language target setting panel.

-msext keyword

  Allow or disallow Microsoft Visual C++ extensions.

  The options for keyword are:

  - on
    Enables these extensions:
    - Redefinition of macros.
Command Line Tools

Compiler Options

- Allows XXX::yyy syntax when declaring method yyy of class XXX.
- Allows extra commas.
- Ignores casts to the same type.
- Treats function types with equivalent parameter lists but different return types as equal.
- Allows pointer-to-integer conversions, and various syntactical differences.

- off
  Disable extensions. Default on non-x86 target platforms.

You can also add #pragma cpp_extensions on|off in a prefix file.

-[no]multibyte[aware]
Enable multi-byte character encoding for source text, comments, and strings.

IDE Equivalent Select “Multi-Byte Aware” in C/C++ Language target setting panel.

-once
Prevent header files from being processed more than once. You can also add #pragma once on in a prefix file.

-pragma pragma_name pragma_setting
Define a pragma for the compiler. For example: -pragma mmx on
You can also add #pragma statement in source code.

-r[quireprotos]
Require function and method prototypes.

IDE Equivalent Select “Require Function Prototypes” in C/C++ Language target setting panel.

-relax_pointers
Relax pointer type-checking rules.

IDE Equivalent Select “Relaxed Pointer Type Rules” in C/C++ Language target setting panel.
**-RTTI on | off**
Select runtime typing information (for C++). Default is on.

IDE Equivalent Select “Enable RTTI” in the C/C++ Language target setting panel.

**-stdkeywords on | off**
Allow only standard keywords. Default is off.

IDE Equivalent Select “ANSI Keywords Only” in C/C++ Language target setting panel.

**-str[ings] keyword[,...]**
Specify string constant options.
The options for keyword are:
- [no]reuse
  Reuse strings. Equivalent strings are the same object. Default.
- [no]pool
  Pool strings into a single data object.

IDE Equivalent Select “Reuse Strings” or “Pool Strings” in C/C++ Language target setting panel.

**-strict on | off**
Specify ANSI strictness checking. Default is off.

IDE Equivalent Select “ANSI Strict” in C/C++ Language target setting panel.

**-trigraphs on | off**
Enable recognition of trigraphs. Default is off.

IDE Equivalent Select “Expand Trigraphs” in C/C++ Language target setting panel.

**-wchar_t on | off**
Enable wchar_t as a built-in C++ data type. Default is off.

IDE Equivalent Select “Enable wchar_t Support” in C/C++ Language target setting panel.
**x86 Code Generation Options**

- **-abi -3dnow**
  Specify the 3D-Now! MMX calling convention. Implies -inst 3dnow

  **IDE Equivalent**
  Choose Edit > targetname Settings... > Code Generation > x86 CodeGen target panel. Enable “Use Multimedia Calling Convention”. Enable “Processor-specific instructions” and enable MMX and 3DNow! checkboxes.

- **-align keyword[,...]**
  Specify structure and array alignment.

  The options for keyword are:
  - 1 | byte | packed
    Byte alignment.
  - 2
    short alignment.
  - 4
    long alignment. Default.
  - 8
    double word (floating double) alignment.
  - 16
    quadword alignment.

  **IDE Equivalent**
  Choose Edit > targetname Settings... > Code Generation > x86 CodeGen target panel. In the Code generation group, select 1 to 16 in the Struct alignment dropdown list.

  - array[members]
    Align array members. You can also use #pragma align_array_members on in a prefix file.

- **-exc[ceptions] keyword**
  Specify exception handling style. Implies -cpp_exceptions on

  The options for keyword are:
  - mw | cw
Command Line Tools
Compiler Options

- **ms | microsoft**
  Use Microsoft-compatible exception handling.

**IDE Equivalent**
Choose Edit > *targetname* Settings... > Code Generation > x86 CodeGen target panel. In the Microsoft Compatibility group, enable C++ exception handling.

- **-inst[ructionset] keyword[...]**
  Specify instruction set extensions.
  The options for *keyword* are:
  - **generic | none**
    Basic 80x86 instruction set. Default.
  - **extended**
    Use extended instructions. Only effective with `-proc pentium` or newer.
  - **mmx**
    Intel MMX extensions. Implies `-inst extended`
  - **fcomi**
    Use FCOMI + Jcc in float comparisons instead of FCOMI + FNSTSW. Implies `-inst extended`.
  - **cmov**
    Use CMOVcc for conditional moves. Implies `-inst extended`.
  - **sse**
    SSE extensions. Implies `-inst extended`
  - **sse2**
    SSE extensions. Implies `-inst extended, sse`
  - **3dnow | 3d**
    AMD K6-3D extensions. Implies `-inst extended`

**IDE Equivalent**
Choose Edit > *targetname* Settings... > Code Generation > x86 CodeGen target panel. Enable Processor-specific instructions and enable appropriate selections.

- **-machinecodelist**
  Generate *.ASM file for each source file.

**IDE Equivalent**
Choose Project > Disassemble
-proc[essor] keyword

Specify the target processor.

The options for keyword are:

- x86
  Generic x86. Default.
- 586 | Pentium
  Pentium processor.
- 686 | PPro | PentiumPro
  Pentium Pro processor.
- PII | Pentium2
  Pentium II processor.
- PIII | Pentium3
  Pentium III processor.
- PIV | Pentium4
  Pentium IV processor.
- AMDK6 | K6
  AMD K6 processor.
- Athlon
  AMD Athlon processor.
- AthlonXP
  AMD Athlon XP processor.

IDE Equivalent
Choose Edit > targetname Settings... > Code Generation > x86 CodeGen target panel. Make selection from Target processor: dropdown list.

-save-temps

Emit machine code listing and preprocessor output in addition to object code.

IDE Equivalent
Choose Project > Disassemble and Project > Preprocess
Code Optimization Options

**NOTE**

All optimization options besides `-opt off | on | all | space | speed | level =...` are for backwards compatibility. You can supersede other optimization options by using `-opt level=xxx`

- **-O**
  Same as `-O2`

- **-O+keyword[,...]**
  Cased. Control optimization. You can combine options, such as `-O4,p`

  The options for `keyword` are:
  - **0**
    Same as `-opt off`
  - **1**
    Same as `-opt level=1`
  - **2**
    Same as `-opt level=2`
  - **3**
    Same as `-opt level=3`
  - **4**
    Same as `-opt level=4, intrinsics`

**IDE Equivalent**

Choose Edit > `targetname` Settings... > Code Generation > Global Optimizations panel and move slider to Optimizations Off.

- **p**
  Same as `-opt speed`

- **s**
  Same as `-opt space`

**IDE Equivalent**

Choose Edit > `targetname` Settings... > Code Generation > Global Optimizations panel and select Optimize For: Faster Execution Speed or Smaller Code Size.
**Command Line Tools**

**Compiler Options**

- **-opt keyword[...]**

Specify code optimization options.

The options for `keyword` are:

- **off | none**
  
  Suppress all optimizations. Default.

- **on**
  
  Same as `-opt level=2`

**IDE Equivalent**

Choose Edit > `targetname` Settings... > Code Generation > Global Optimizations panel and move slider to Level 2.

- **all | full**
  
  Same as `-opt speed,level=4,intrinsics, noframe`

**IDE Equivalent**

Choose Edit > `targetname` Settings... > Code Generation > Global Optimizations panel and move slider to Level 4.

**IDE Equivalent**

Choose Edit > `targetname` Settings... > Code Generation > x86 CodeGen panel and enable Inline intrinsic functions checkbox.

- **[no]space**
  
  Optimize for space.

- **[no]speed**
  
  Optimize for speed.

**IDE Equivalent**

Choose Edit > `targetname` Settings... > Code Generation > Global Optimizations panel and select Optimize For: Faster Execution Speed or Smaller Code Size.

- **l[evel]=num**

  Set optimization level.

  The options for `num` are:

  - **0**
    
    Global register allocation only for temporary values.

  - **1**
    
    Adds dead code elimination, branch and arithmetic optimizations, expression simplification, and peephole optimization.
Command Line Tools

Compiler Options

- 2
  Adds common subexpression elimination, copy and expression
  propagation, stack frame compression, stack alignment, and fast
  floating-point to integer conversions.

- 3
  Adds dead store elimination, live range splitting, loop-invariant
  code motion, strength reduction, loop transformations, loop
  unrolling (with -opt speed only), loop vectorization,
  lifetime-based register allocation, and instruction scheduling.

- 4
  Like level 3, but with more comprehensive optimizations from
  levels 1 and 2.

IDE Equivalent

Choose Edit > targetname Settings... > Code Generation > Global
Optimizations panel and move slider to Optimizations Off or Level 1 to 4,
as needed.

NOTE

For num options 0 through 4 inclusive, the default is 0.

- [no]cse | [no]commons

  Common subexpression elimination. You can also add
  #pragma opt_common_subs on|off to a prefix file.

- [no]deadcode

  Removal of dead code. You can also add #pragma
  opt_dead_code on|off to a prefix file.

- [no]deadstore

  Removal of dead assignments. You can also add #pragma
  opt_dead_assignments on|off to a prefix file.

- [no]lifetimes

  Computation of variable lifetimes. You can also add #pragma
  opt_lifetimes on|off to a prefix file.

- [no]loop[invariants]

  Removal of loop invariants. You can also add #pragma
  opt_loop_invariants on|off to a prefix file.

- [no]propagation

  Common subexpression elimination. You can also add #pragma
  opt_common_subs on|off to a prefix file.
Propagation of constant and copy assignments. You can also add `#pragma opt_propagation on|off` to a prefix file.

- `[no]strength`
  Strength reduction. Reducing multiplication by an array index variable to addition. You can also add `#pragma opt_strength_reduction on|off` to a prefix file.

- `[no]dead`
  Same as `-opt [no]deadcode` and `[no]deadstore`. You can also add `#pragma opt_dead_code on|off` and `#pragma opt_dead_assignments on|off` to a prefix file.

- `[no]peep[hole]`
  Peephole optimization. You can also add `#pragma peephole on|off` to a prefix file.

- `[no]color[ing]`
  Register coloring. You can also add `#pragma register_coloring on|off` to a prefix file.

- `[no]intrinsics`
  Inlining of intrinsic functions.

**IDE Equivalent**

Choose Edit > `targetname` Settings... > Code Generation > x86 CodeGen panel and enable the Inline intrinsic functions checkbox.

- `[no]schedule`
  Perform instruction scheduling.

- `display | dump`
  Display complete list of active optimizations.

**C/C++ Warning Options**

-`-w[arn[ings]]` `keyword[,...]`

Global.

The options for `keyword` are:

- `off`
Passed to all tools. Turn off all warnings.

- **on**
  Passed to all tools. Turn on most warnings.

**IDE Equivalent**
Choose Edit > `targetname` Settings... > Language Settings > C/C++ Warnings

- **[no]cmdline**
  Passed to all tools. Command line driver/parser warnings.

- **[no]err[or] | [no]iserr[or]**
  Passed to all tools. Treat warnings as errors.

**IDE Equivalent**
Choose Edit > `targetname` Settings... > Language Settings > C/C++ Warnings panel and enable the Treat All Warnings As Errors checkbox.

- **all**
  Turn on all warnings. Requires prototypes.

**IDE Equivalent**
Choose Edit > `targetname` Settings... > Language Settings > C/C++ Warnings panel and enable the Treat All Warnings As Errors checkbox. Then select the C/C++ Language panel and enable the Require Function Prototypes checkbox.

- **[no]pragmas | [no]illpragmas**
  Illegal pragmas.

**IDE Equivalent**
Choose Edit > `targetname` Settings... > Language Settings > C/C++ Warnings panel and enable Illegal Pragmas.

- **[no]empty[decl]**
  Empty declarations.

**IDE Equivalent**
Choose Edit > `targetname` Settings... > Language Settings > C/C++ Warnings panel and enable Empty Declarations.

- **[no]possible | [no]unwanted**
  Possible unwanted effects.

**IDE Equivalent**
Choose Edit > `targetname` Settings... > Language Settings > C/C++ Warnings panel and enable Possible Errors.

- **[no]unusedarg**
Unused arguments.

IDE Equivalent: Choose Edit > targetname Settings... > Language Settings > C/C++ Warnings panel and enable Unused Arguments.

- [no]unusedvar
  Unused variables.

IDE Equivalent: Choose Edit > targetname Settings... > Language Settings > C/C++ Warnings panel and enable Unused Variables.

- [no]unused
  Same as -w [no]unusedarg, [no]unusedvar
- [no]extracomma | [no]comma
  Extra commas

IDE Equivalent: Choose Edit > targetname Settings... > Language Settings > C/C++ Warnings panel and enable Extra Commas.

- [no]pedantic | [no]extended
  Pedantic error checking.

IDE Equivalent: Choose Edit > targetname Settings... > Language Settings > C/C++ Warnings panel and enable Extended Error Checking.

- [no]hidevirtual | [no]hidden[virtual]
  Hidden virtual functions.

IDE Equivalent: Choose Edit > targetname Settings... > Language Settings > C/C++ Warnings panel and enable Hidden Virtual Functions.

- [no]implicit[conv]
  Implicit arithmetic conversions.

IDE Equivalent: Choose Edit > targetname Settings... > Language Settings > C/C++ Warnings panel and enable Implicit Arithmetic Conversions.

- [no]notinlined
  inline functions not inlined

IDE Equivalent: Choose Edit > targetname Settings... > Language Settings > C/C++ Warnings panel and enable Non-Inlined Functions.
Command Line Tools

Compiler Options

- [no]largeargs
  Passing large arguments to unprototyped functions. You can also add #pragma warn_largeargs on|off to a prefix file.

- [no]structclass
  Inconsistent use of class and struct.

IDE Equivalent
Choose Edit > targetname Settings... > Language Settings > C/C++ Warnings panel and enable Inconsistent Use of class and struct Keywords.

- [no]padding
  Padding added between structure members. You can also add #pragma warn_padding on|off to a prefix file.

- [no]notused
  Function return value not used. You can also add #pragma warn_resultnotused on|off to a prefix file.

- [no]unusedexpr
  Use of expressions as statements without side effects. You can also add #pragma warn_no_side_effect on|off to prefix file.

- display | dump
  Display list of active warnings.

Debugging Control Options

-g
Global. Cased. Generate symbolic debugging information. Same as -sym on

-sym keyword[,...]
Global. Specify debugging options.

The options for keyword are:

- off - Do not generate symbolic debugging information. Default.

IDE Equivalent
In Debug column of Project window, select dot to remove it.

TNW–148 Targeting the NetWare® Operating System, PDK 5.0
- on
  Turn on symbolic debugging information.

IDE Equivalent
  In Debug column of Project window, click on the area aligned with the
desired source file.

- [no]codeview
  Store symbolic debugging information in CodeView 5.0 format.
  Default.

IDE Equivalent
  Choose Edit > targetname Settings... > Linker > NLM Linker panel and
  enable the Generate CV SYM File checkbox.

- [no]internal
  Store internal debugging information.

IDE Equivalent
  Choose Edit > targetname Settings... > Linker > NLM Linker panel and
  enable the Generate Internal Debugger Records checkbox.

Linker Options

- General Options
- Netware x86 Linker Options
- Netware Project Options
- Debugging Control Options
- Warning Options
- Disassembler Options

General Options

-dis[assemble]
  Global. Disassemble object code and do not link. Implies -nostdlib

IDE Equivalent
  Choose Project > Disassemble. The IDE always sends output to a new
  unsaved editor window.

-L+ l -I path
  Global. Cased. Add the specified library search path. The default is to
  search the current working directory and then system directories (see –
defaults). Search paths have global scope over the command line and are searched in the order specified.

**IDE Equivalent**
Choose Edit > targetname Settings... > Access Paths and add non-recursive access paths.

**-lr path**
Global. Similar to `-l`, but adds a recursive library search path.

**IDE Equivalent**
Choose Edit > targetname Settings... > Access Paths and add recursive paths.

**-l+file**
Cased. Add a library by searching access paths for file named `lib<file>.<ext>` where `<ext>` is a typical library file name extension, and added before system libraries.

See `-defaults`

**IDE Equivalent**
Choose Project > Add Files... to add a file to the project.

**-[no]defaults**
Global. Same as `- [no] stdinc`. Default. The command line tool uses additional environment variables to provide "defaults", but in the IDE, all access paths and libraries are explicit.

**-nofail**
Continue importing or disassembling despite errors in earlier files.

**-[no]stdlib**
Global. Use system library access paths (specified by the environment variable `%MWLibraries%`) and add system libraries (specified by the environment variables `%MWLibraryFiles%`). Default. The command line tool uses additional environment variables to provide "defaults", but in the IDE, all access paths and libraries are explicit.

**-S**
Global. Cased. Disassemble and send output to file. Do not link. Implies `-nostdlib`

**IDE Equivalent**
Choose Project > Disassemble and save output to a file.
Netware x86 Linker Options

-check symbol
Set NLM check-before-unload routine. For 'symbol', maximum length is 63 characters. Default is none.

IDE Equivalent
Choose Edit > targetname Settings... > Linker > NLM Linker panel and enter symbol in the Entry Points > Check: text box.

-commandfile filename
Add a command file (named *.def or *.exp) for the linker.

IDE Equivalent
Add *.def, *.imp or *.exp file to Project window.

-entry symbol
Set NLM entry point. For 'symbol', maximum length is 63 characters. Default is _Prelude.

IDE Equivalent
Choose Edit > targetname Settings... > Linker > NLM Linker panel and enter symbol in Entry Points > Start: text box.

-exit symbol
Set NLM exit routine. For 'symbol', maximum length is 63 characters. Default is _Stop.

IDE Equivalent
Choose Edit > targetname Settings... > Linker > NLM Linker panel and enter symbol in Entry Points > Exit: text box.

-flags keyword[...] Specify NLM flags
• reent[rant]
• mult[iple]
• synch[ronize]
• pseudo[preemption]
• os[domain]
• autounload

IDE Equivalent
Choose Edit > targetname Settings... > Linker > NLM Linker panel and select from Flags dropdown list.
**Command Line Tools**

**Linker Options**

- **-map filename**
  
  Generate a link map file. The default file name is `<outfile>.MAP`.

  **IDE Equivalent**
  
  Choose Edit > *targetname* Settings... > Linker > NLM Linker panel and enable Generate Link Map.

- **-nocheck**
  
  Do not use a check routine; default.

  **IDE Equivalent**
  
  Choose Edit > *targetname* Settings... > Linker > NLM Linker panel and leave Entry Points > Check: text box blank.

- **-o file**
  
  Specify output file.

  **IDE Equivalent**
  
  Choose Edit > *targetname* Settings... > Target > NLM Target and specify filename in Output File: text box.

- **-[no]zerobss**
  
  Create a zero-initialized section for .bss data.

  **IDE Equivalent**
  
  Choose Edit > *targetname* Settings... > Linker > NLM Linker and enable the Allocate Uninitialized Data in File checkbox.

**Netware Project Options**

- **-copy[right] string**
  
  Set copyright string. For 'string', maximum length is 255 characters. Default is none.

  **IDE Equivalent**
  
  Choose Edit > *targetname* Settings... > Target > NLM Target and enter string in Copyright text box.

- **-desc[ription] string**
  
  Set NLM description. For 'string', maximum length is 127 characters. Default is 'no description'.

  **IDE Equivalent**
  
  Choose Edit > *targetname* Settings... > Target > NLM Target and enter string in Description text box.
-nlmversion

Specify version. Default is 1,0,0 (major, minor, revision).

IDE Equivalent
Choose Edit > targetname Settings... > Target > NLM Target and enter information in Version group.

-screename string

Specify screen name. For 'string', maximum length is 71 characters. Default is 'DEFAULT'.

IDE Equivalent
Choose Edit > targetname Settings... > Target > NLM Target and select User Specified from the Screen Name: dropdown list and enter string in the Initial Screen Name text box.

-stacksize size

Specify size of stack in bytes. Default is 8192.

IDE Equivalent
Choose Edit > targetname Settings... > Target > NLM Target and enter size in Stack Size text box.

-threadname string

Specify thread name. For 'string', maximum length is 17 characters. Default is none.

IDE Equivalent
Choose Edit > targetname Settings... > Target > NLM Target and enter string in Initial Thread Name text box.

-type keyword

Keywords are:

- library
- generic - generic NLM; default
- LAN - LAN driver
- disk - disk driver
- namespace - name space support
- utility
- mirror - mirrored server link
- OS - operating system NLM
- OS_high - OS NLM, paged high
Command Line Tools

Linker Options

- **host** - host adapter
- **custom** - custom device

**IDE Equivalent**
Choose Edit > **targetname** Settings... > Target > NLM Target and select from Output Type dropdown list.

**Debugging Control Options**

**-g**
Global. Cased. Generate debugging information. Same as `-sym on`

**-sym keyword**, ...]
Global. Specify debugging options.
The options for `keyword` are:
- **off**
  Do not generate symbolic debugging information. Default.
- **on**
  Generate symbolic debugging information.

**IDE Equivalent**
Choose Edit > **targetname** Settings... > Linker > NLM Linker and enable or disable the Generate CV SYM File checkbox.

- **[no]codeview**
  Store symbolic debugging information in CodeView 5.0 format. Default.
- **[no]internal**
  Store internal debugging information.

**IDE Equivalent**
Choose Edit > **targetname** Settings... > Linker > NLM Linker and enable the Generate Internal Debugger Records checkbox.

**-osym filename**
For command line linker. Write symbolic file information to `filename`. Implies `-sym on`. The default is `<outfile>.NCV`. 
Warning Options

-warn[ings] keyword[,...]
Global. Warning options.

The options for keyword are:

- **off**
  Turn off all warnings.

- **on**
  Turn on most warnings.

- **[no]cmdline**
  Command line driver/parser warnings.

- **[no]err[or] | [no]iserr[or]**
  Treat warnings as errors.

- **display | dump**
  Display list of active warnings.

Disassembler Options

-show keyword[,...]
Specify disassembly options.

The options for keyword are:

- **only | none**
  As in -show none or -show only, code, data.

- **all**
  Show everything.

IDE Equivalent

Choose Edit > targetname Settings... > Linker > x86 Disassembler panel.

- **[no]code | [no]text**
  Show disassembly of code sections. Default.

IDE Equivalent

Choose Edit > targetname Settings... > Linker > x86 Disassembler panel and enable Code sections in Decoding Options group.

- **[no]comments**
Command Line Tools

Linker Options

Show comments showing jump targets and decimal values.

**IDE Equivalent**
Choose Edit > targetname Settings... > Linker > x86 Disassembler panel and enable Show comments in Code Disassembly Options group.

- **[no]** `data`
  Show data sections. Default.

- **[no]** `debug`
  Show debugging sections.

- **[no]** `exceptions`
  Show exception tables.

- **[no]** `headers`
  Show object headers. Default.

- **[no]** `sheaders`
  Show section headers. Default.

**IDE Equivalent**
Choose Edit > targetname Settings... > Linker > x86 Disassembler panel and enable the related checkbox in Decoding Options group.

- **[no]** `hex`
  Show addresses and opcodes in code disassembly. Implies -show code. Default.

**IDE Equivalent**
Choose Edit > targetname Settings... > Linker > x86 Disassembler panel and enable Show address and opcodes in Code Disassembly Options group.

- **[no]** `names`
  Show symbol and string tables and definitions. Default.

**IDE Equivalent**
Choose Edit > targetname Settings... > Linker > x86 Disassembler panel and enable Show symbol and string tables in Decoding Options group. Also enable Show symbol definitions in General Options group.

- **[no]** `defs`
  Show symbol definition labels in code and data. Default. In the IDE, enable Show symbol definitions in General Options group of x86 Disassembler panel.

- **[no]** `syms | symbols`

---

TNW–156  Targeting the NetWare® Operating System, PDK 5.0
Show symbol locations in code and data. Default.

**IDE Equivalent**
Choose Edit > *targetname* Settings... > Linker > x86 Disassembler panel and enable Symbol and string tables in Decoding Options group.

- **[no]** _raw_
  Show undisplayed sections as hexadecimal dumps.

**IDE Equivalent**
Choose Edit > *targetname* Settings... > Linker > x86 Disassembler panel and enable Display hex dumps in Decoding Options group.

- **[no]** _allraw_
  Show all sections as hexadecimal dumps.

**IDE Equivalent**
Choose Edit > *targetname* Settings... > Linker > x86 Disassembler panel and enable Display hex dumps and Dump undecoded sections in Decoding Options group.

- **[no]** _relocs_
  Show relocation tables and resolve relocatable symbol references.

**IDE Equivalent**
Choose Edit > *targetname* Settings... > Linker > x86 Disassembler panel and enable Relocation tables in the Decoding Options group and Resolve relocations in the General Options group.

- **[no]** _reloctabs_
  Show relocation tables.

- **[no]** _source_
  Show source in disassembly (for objects compiled with debugging information only). Implies -show code. Default.

**IDE Equivalent**
Choose Edit > *targetname* Settings... > Linker > x86 Disassembler panel and enable Interleave source in the Code Disassembly Options group.

- **[no]** _unmangled_
  Show unmangled symbols.

**IDE Equivalent**
Choose Edit > *targetname* Settings... > Linker > x86 Disassembler panel and enable Unmangle symbols in the General Options group.

- **[no]** _verbose_
### Command Line Tools

#### Linker Options

<table>
<thead>
<tr>
<th>IDE Equivalent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Show verbose information. If <code>-show raw</code> is specified, show displayed sections as hex dumps. If <code>-show names</code> is specified, show table of unmangled symbols and auxiliary symbols.</td>
</tr>
<tr>
<td>Choose Edit &gt; <code>targetname</code> Settings... &gt; Linker &gt; x86 Disassembler panel and enable Verbose Information in the General Options group.</td>
<td></td>
</tr>
</tbody>
</table>
The NetWare/x86 Debugger

This chapter explains features that are unique to the NetWare®/x86 implementation of the CodeWarrior™ debugger.

Refer to online help or the CodeWarrior™ IDE User Guide for general instructions on using the CodeWarrior debugger.

This chapter contains these sections:
- Setting Up for Client/Server Debugging
- Project Settings for Debugging
- The Runtime Settings Target Settings Panel
- Tutorial: Debugging an NLM
- The Processes Debugger Window
- Debugging Multiple NLMs
- Debugging a Static Library
- Debugging NLMs Created with Third Party Tools

Setting Up for Client/Server Debugging

The NetWare/x86 debugger is a client/server debugger. The debugger runs on a Windows® PC. The NLM being debugged runs on a NetWare server.

While debugging, the PC and server communicate over one of two kinds of links:
- An asynchronous serial communications link.
- An Ethernet local area network (LAN) link that uses the Transport Control Protocol/Internet Protocol (TCP/IP) protocol.
These topics explain how to set up a serial link and an Ethernet-TCP/IP link:

- Setting Up a Serial Link
- Setting Up an Ethernet-TCP/IP Link

**Setting Up a Serial Link**

To set up a serial link between the PC and the NetWare server, follow these steps:

1. Connect a null modem cable between a serial port on the PC and a serial port on the NetWare server.
2. Start the CodeWarrior IDE.
3. Configure the debugger on the Windows PC.

To do this, follow these steps:

a. Select **Edit > Preferences**.
   
   The IDE Preferences window appears.

b. Select Global Connection... in the list IDE Preference Panels (Debugger group).

   The Global Connection... panel appears on the right side.

c. Select View Serial Settings from the View Connection Type: dropdown menu.

   The serial communications options appear.

   **Figure 9.1** shows the Global Connection... preference panel with the serial options visible.
For the Port: item, select the name of the port to which you connected the null modem cable. Your choice must be COM1, COM2, COM3, or COM4.

For the Rate: item, select the speed (in bits per second) at which you want the PC and the NetWare server to communicate. Your choice must be between 2,400 bps and 115,200 bps, inclusive.

**NOTE** Make sure your PC supports the selected port and rate.

For the Data Bits: item, you must select 8 data bits.

For the Parity: item, you must select none.

For the Stop Bits: item, you must select 1 stop bit.

For the Flow Control: item, you must select none.
The NetWare/x86 Debugger
Setting Up for Client/Server Debugging

j. Click OK.
   The IDE Preferences window closes.

The Windows PC and NetWare server are now set up to communicate over a serial link.

Setting Up an Ethernet-TCP/IP Link

To set up an Ethernet-TCP/IP link between the PC and NetWare server, follow these steps:

1. Install an Ethernet card in the Windows PC.
   To do this, follow the instructions that came with the card.

2. Install and configure TCP/IP software on the Windows PC.
   To do this, follow the instructions in your Windows operating system documentation.

3. Install an NE2000 compatible Ethernet card in the NetWare server.
   To do this, follow the instructions that came with the NE2000 compatible card.

   NOTE
   The NetWare server must have an NE2000 compatible Ethernet card dedicated to communications with the debugger on the PC.
   If desired, you can install a second NE2000 card in the server for other LAN communication.

4. Install and configure TCP/IP software on the NetWare server.
   To do this, follow the instructions in your NetWare operating system documentation.

5. Connect an Ethernet cable to the Ethernet cards in the PC and NetWare server.

6. Obtain unused TCP/IP address to use for debugging.
   Ask your LAN Administrator for this address.

7. Start the CodeWarrior IDE

8. Configure the debugger on the Windows PC
   a. Select Edit > Preferences - IDE Preferences window appears.
b. Select Global Connection... in the list of IDE Preference Panels. The Global Connection... panel appears on the right side.

c. Select View TCP/IP Settings from the View Connection Type dropdown menu. The TCP/IP settings options appear. Figure 9.2 shows the Global Connection... preference panel with the TCP/IP option visible.

**Figure 9.2** The Global Connection Panel Showing the TCP/IP Options

<table>
<thead>
<tr>
<th>IDE Preferences</th>
<th>Global Connection Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDE Preference Panels</td>
<td>Global Connection Settings</td>
</tr>
<tr>
<td>General</td>
<td>View Connection Type</td>
</tr>
<tr>
<td>Build Settings</td>
<td>View TCP/IP Settings</td>
</tr>
<tr>
<td>IDE Extras</td>
<td>Configure connection settings here. Choose the connection type in the Debugger Target Settings panel.</td>
</tr>
<tr>
<td>Plugin Settings</td>
<td>Host Name: 1.1.1.1</td>
</tr>
<tr>
<td>Shielded Folders</td>
<td></td>
</tr>
<tr>
<td>Source Trees</td>
<td></td>
</tr>
<tr>
<td>Editor</td>
<td></td>
</tr>
<tr>
<td>Code Completion</td>
<td></td>
</tr>
<tr>
<td>Editor Settings</td>
<td></td>
</tr>
<tr>
<td>Font &amp; Tab</td>
<td></td>
</tr>
<tr>
<td>Text Colors</td>
<td></td>
</tr>
<tr>
<td>Debugger</td>
<td></td>
</tr>
<tr>
<td>Display Settings</td>
<td></td>
</tr>
<tr>
<td>Windowing</td>
<td></td>
</tr>
<tr>
<td>Global Settings</td>
<td></td>
</tr>
<tr>
<td>Remote Connections</td>
<td></td>
</tr>
<tr>
<td>Global Connection...</td>
<td></td>
</tr>
<tr>
<td>Java Settings</td>
<td></td>
</tr>
</tbody>
</table>

- Factory Settings
- Revert
- Import Panel
- Export Panel

- OK
- Cancel
- Apply

d. Enter the unused TCP/IP address for the Host Name:
e. Click OK - IDE Preferences window closes

The Windows PC and NetWare server are now set up to communicate using TCP/IP over Ethernet.
Project Settings for Debugging

Each NLM that you want to debug must include symbolic debugging information and be in CodeView format.

**NOTE**
The NetWare/x86 debugger cannot use symbolic debugging information in SYM format.

The NetWare/x86 linker puts your NLM’s CodeView symbolic information in a file that ends with the suffix `.NCV`.

All projects you create using the stationery included in the CodeWarrior Add-on Tools for NetWare includes a debug build target. A debug build target is a build target for which all settings needed to debug your project are correctly set.

If you did not create your project using stationery, you must configure your project so that it generates CodeView symbolic debugging information. To do this, follow these steps:

1. Start the CodeWarrior IDE.
2. Open your project.
3. Select your project’s debug build target.

To do this, follow these steps:

a. In the project window, open the build target dropdown menu.

   *Figure 9.3* shows a project window with the build target dropdown menu open.

*Figure 9.3* Project Window with Build Target Dropdown Menu
b. Select the name of your debug build target

c. Press Enter or click the left mouse button.

The project manager sets the selected build target as the active build target.

**NOTE**

It is common to create at least two build targets, one set to create a release version and another set to create a debug version of your software.

Refer to the *CodeWarrior™ IDE User Guide* or online help for instructions on how to add a build target to a project.

4. Display the Target Settings window for your project.

See “Target Settings Overview” on page 37 for instructions that explain how to display and use the Target Settings window.

5. Display the NLM Linker panel in the Target Settings panels.

6. Enable the Generate CodeView Debug info check box.

7. Enable the Generate Internal Debugger Records check box.

*Figure 9.4* shows the NLM Linker target settings panel with the Generate CodeView Debug info and Generate Internal Debugger Records check boxes enabled.
8. Click OK.

The selected build target is now set up so that you can debug it using the NetWare/x86 debugger.

**The Runtime Settings Target Settings Panel**

Before you debug an NLM, you may need to make an entry in the Runtime Settings panel.

The NetWare/x86 debugger uses one item on the Runtime Settings panel, the Program Arguments: text box. This text box is part of the General Settings group of items.

If your NLM accepts command line arguments, you can enter them in the Program Arguments: text box. The debugger passes the
arguments to the NLM’s `main()` function at the beginning of each debug session.

**NOTE** Refer to Table 4.3 for more information about the Runtime Settings panel. For more information about NLM Load arguments, refer to the Novell documentation.

To display the Runtime Settings panel, follow these steps:

1. Start the CodeWarrior IDE.
2. Open your project.
3. Select Edit > `targetname` Settings... (ALT-F7) to display Target Settings Panels.
4. Select Runtime Settings

**Figure 9.5** shows the Runtime Settings panel.

![Runtime Settings Target Setting Panel](image-url)
Tutorial: Debugging an NLM

This section presents a tutorial that demonstrates how to debug a simple NLM.

To complete the tutorial, follow these steps:

1. Start the CodeWarrior IDE.
2. Create a new project based on the Netware Generic C stationery.
   a. Select File > New - the New dialog box appears
   b. Click the Project tab in the New dialog box.
      Figure 9.6 shows the New dialog box with the Project options displayed.

   c. Select NetWare Stationery from the list on the left side of the New dialog box.
d. Enter hello in the Project name: text box.
e. Enter C:\hello in the Location: text box.
f. Click OK.

The New Project dialog box appears. See Figure 9.7.

Figure 9.7 The New Project Dialog Box

![Figure 9.7 The New Project Dialog Box](image)

g. Select Generic NLM C from the Project Stationery list in the New Project dialog box.

h. Click OK.

The CodeWarrior IDE:
- Creates a new project based on the Generic C stationery.
- Names the project hello.mcp.
- Displays the project window shown in Figure 9.8.
3. Select the Generic C Debug build target.
   a. In the project window, open the build target dropdown menu. 
      Figure 9.9 shows a project window with the build target dropdown menu open.

   b. Select Generic C Debug
   c. Press Enter or click the left mouse button.

      The project manager sets the Generic C Debug build target as the active build target.

4. Select Edit > Preferences.
   The IDE Preferences window appears.
5. Create a serial link or an Ethernet-TCP/IP link between the PC and the NetWare server.

Refer to Setting Up a Serial Link for instructions that explain how to set up a serial link.

Refer to Setting Up an Ethernet-TCP/IP Link for instructions that explain how to set up an Ethernet-TCP/IP link.

6. Select Edit > targetname Settings... (ALT-F7) to display the Target Settings window.

7. Select the NLM Target panel.

8. Select Console from the Screen Name: dropdown menu in the NLM Target settings panel. The NLM output will appear on the console of the NetWare server.

Figure 9.10 shows the NLM Target panel with Console selected for Screen Name:

Figure 9.10  NLM Target Settings Panel
9. Click Apply to accept settings.

10. Select the Mpkxdc Panel

Use the Mpkxdc panel to define settings for the Novell Mpkxdc utility and specify a remote target directory on the Netware server in which to download your NLM program. More information for this panel is provided in the Mpkxdc Panel section of chapter 4.

Type or click Choose button to locate the target folder path where NLM files are copied for debugging on the NetWare server. After clicking Choose button, select Absolute Path as the path type. When you build (make) your NLM project in the CodeWarrior IDE, the NLM file is downloaded to the path specified.

11. Click OK

Figure 9.11 shows the Mpkxdc panel.

Figure 9.11 Mpkxdc Panel
12. Select Project > Make (F7) to build your NLM and download hellod.nlm from your PC to the target folder path on the NetWare server. This file is the NLM you just built. It is in the directory C:\hello on your PC.

13. Select Project > Debug to enable the NetWare/x86 debugger.

**NOTE** Do not delete the file hellod.NCV from the PC.

This file contains your NLM’s CodeView symbolic debugging information. The debugger (which runs on the PC) needs the information in this file.

14. At the NetWare server’s console, press:

```
LEFT-SHIFT + RIGHT-SHIFT + ALT + ESC
```

The NetWare server enters debug mode. You can tell the server is in debug mode because it displays a # prompt.

15. At the NetWare server’s # prompt, enter one of the commands listed below.

a. If you are using a serial communications link for debugging, enter this command:

```
rdebug serial com_port baud baud_rate port port_address
```

   - Replace `com_port` with the name of the serial port you are using for debugging on the server.
   
   The value for `com_port` must be COM1, COM2, COM3, or COM4.
   
   - Replace `baud_rate` with the same value you entered for Rate: in the Global Connection... preference panel on the PC.
   
   - Replace `port_address` with the address of the serial port you entered for `com_port`.

b. If you are using an Ethernet-TCP/IP communications link for debugging, enter this command:

```
rdebug ethernet_card_name ip unused_tcp/ip_address port=x
```

   - Replace `ethernet_card_name` with the name you gave to the debug Ethernet card when you installed it in the server.
The NetWare/x86 Debugger
Tutorial: Debugging an NLM

- Replace `unused_tcp/ip_address` with the same TCP/IP address you entered in the Global Connection... preference panel on the PC.
- Replace `x` with the port specified in the documentation that came with the debug Ethernet card you installed in the server.

16. At the NetWare server’s console, press Enter.

   The NetWare remote debugger activates.

   A red line containing the text “Remote Debugger Active Abort <esc>” appears at the top of the server’s console.

17. On the PC, press F5.

   The debugger runs and:
   - Displays its main window and log window.
   - Instructs the NetWare server to load your NLM.
   - Halts execution of your NLM at the first instruction in the NLM’s `main()` function.
   - Displays the source code for your NLM’s `main()` function in its main window.

   Figure 9.12 shows the debugger’s main window.
Figure 9.12  The Debugger’s Main Window

Figure 9.13  The Debugger’s Log Window.

Figure 9.13  The Debugger’s Log Window
NOTE Refer to online help or the CodeWarrior IDE User Guide for more debugging information. You can set Breakpoints and Watchpoints in the Debugger’s main window. You can also edit memory data and change the value of a register or its condition code.

18. Select Debug > Step Over (F10) to execute one line of code.
19. Press F10 several times until the debugger executes the printf() function in your NLM’s main() function.
   The NetWare console displays the message:
   Hello world from Metrowerks and Novell:
20. Select Project > Debug (F5) to execute the rest of your NLM and exit.
   This ends the debug session on the PC.
21. At the NetWare server’s console, press ESC.
   The NetWare remote debugger halts.
   The red line disappears from the top of the console screen.
22. At the NetWare server’s console, enter g at the # prompt and press Enter. The NetWare server exits debug mode.
   This ends the debug session on the NetWare server.
   You have completed the tutorial.

Monitoring Program Execution

For viewing, setting, and clearing conditional and temporary Breakpoints, refer to online help by selecting the Help > CodeWarrior Help menu option or the IDE User Guide by selecting the Help > Online Manuals menu option.

For viewing, setting, clearing, and setting conditional Watchpoints, also refer to online help or the IDE User Guide. You can also find information for viewing and editing memory, or viewing, editing registers and variables.

The Processes Debugger Window

The NetWare/x86 debugger lets you view the NetWare server’s process spaces and the tasks within each process space.
To view the NetWare server's process spaces and tasks, follow these steps:

1. Debug an NLM.
   To do this, follow the steps in Tutorial: Debugging an NLM.

2. Select View > Processes.
   The Processes window (Figure 9.14) appears.

3. Select NetWare Server from the pull-down menu in the top-left of the processes window.
   The OS address space process appears in the Process pane.

4. Select the OS address space process.

5. Press Enter
   A list of tasks running on the NetWare server appears in the Task pane of the Processes window.

**Debugging Multiple NLMs**

You can use the NetWare/x86 debugger to debug code in other NLMs, which are referenced by your main NLM.

To do this, you must set up the projects as follows:

- [Create Project that Builds the Called NLM](#)
- [Create Project that Builds the Caller NLM](#)
Create Project that Builds the Called NLM

1. Start the CodeWarrior IDE.
2. Open the project that builds the NLM that is to be called (for example, called.mcp).
3. Open the project’s export file (Figure 9.16) or create a new one.

Figure 9.16 Export File

NOTE The export file is a text file with an .EXP extension. The root part of the file name can be anything.

4. In the export file, add the names of all functions and data referenced in your caller project, and defined in the called NLM project.

For example, let’s assume a function named called_function is in a caller.c source file in the caller.mcp project. The function
that this function call references is defined in the `called.c` file in the called.mcp project.

**Figure 9.17 Referencing Functions**

5. Save the export file.
6. If you created a new export file, add it to the called NLM project.
7. Repeat steps 2 through 5 for each called NLM project you want to debug.

**Create Project that Builds the Caller NLM**

1. Start the CodeWarrior IDE.
2. Open the project that builds the calling NLM (for example, `caller.mcp`).
3. Open the project’s import file or create a new one.

**Figure 9.18 Import File**
The NetWare/x86 Debugger

Debugging Multiple NLMs

NOTE

The name of an import file must end in the suffix .IMP. The root part of the file name can be anything.

4. In the import file, add the names of all functions and data making calls from your caller project. The functions are defined in the called NLM projects.

5. Save the import file.

NOTE

You can use one import file to list the names of functions and data defined in several called NLMs.

6. If you created a new import file, add it to the NLM project making the calls (caller.mcp).

7. Edit the linker command file in the caller project, or create a new one.
The NetWare/x86 Debugger
Debugging Multiple NLMs

Figure 9.19 Linker Command File

```
# Sample .DEF file

MODULE called.nlm
# MODULE c.lib.nlm
# Specifies the modules that must be loaded before this NLM is loaded.
# These modules will be loaded automatically when this NLM is loaded.
# In addition to this list, with CodeWarrior for NetWare any NLM modules
# added to the project are also added to the autoload list.
# Example: MODULE c.lib.nlm dapi.nlm

# IMPORT symbol list
# IMPORT @symbol list file
# Specifies a list, or a file containing a list (import file) of symbols
# which will be externally resolved at runtime.
# In addition to this list, with CodeWarrior for NetWare there are two
# other methods for specifying import symbols:
# 1 - Add unpacked NLM modules to the project. All its symbols
#     will be added to this list.
# 2 - Add Import files to the project
# Example: IMPORT ConsolePrintf scanf
# Example: IMPORT @C:\nwsdk\imports\c.lib.imp

# EXPORT symbol list
# EXPORT @symbol list file
# Specifies a list, or a file containing a list of symbols
# which will be exported at runtime.
# Example: EXPORT MyConsolePrintf MyScannf
# Example: EXPORT @C:\myexp.exp

# DATE month, day, year
# Allows the Date in the NLM header to be manually specified.
# By default this date is the date the NLM was built.
# Example: DATE 7, 23, 1998
```

NOTE

A linker command file must end with the extension .DEF.

8. Add the called NLM to the linker command file. For example, enter
   MODULE called.nlm as shown in Figure 9.19. For more
   information about the MODULE command, refer to MODULE.

9. Save the linker command file.

10. If you created a new linker command file, add it to the NLM project
    making the calls (caller.mcp).
NOTE In the project window, select a debug build target from the list of targets. Refer to The Project Manager for information on build targets.

11. Select Edit > targetname Settings... (ALT-F7) to display Target Settings window.

12. Select Other Executables from the list of panels. Figure 9.20 shows the Other Executables panel.

**Figure 9.20** Other Executables Panel

13. In the Other Executables panel, add the path and name of each called NLM you want to debug.

   To do this, follow these steps:

   a. Click Add.

      The Debug Additional Executable dialog box Figure 9.21 appears.
b. Enter the path and name of a called NLM in the File location text box.

**NOTE** The path you enter must be the path to the called NLM on your PC, not to the copy of the NLM on the NetWare server.

c. Do not enable the “Download file during remote debugging” check box.
   The NetWare/x86 debugger does not use this feature.

d. Do not enable the “Debug merged executable” check box.
   The NetWare/x86 debugger does not use this feature.

e. Click OK.
   The file name you entered appears in the File list of the Other Executable panel.

f. Repeat steps a. through e. for each called NLM you want to debug.
The NetWare/x86 Debugger

Debugging a Static Library

You can debug code linked into your NLM from a static library as long as the project settings for the library are set; as defined in Project Settings for Debugging.

To debug code linked from a third party static library, the library must meet the criteria defined in “Debugging NLMs Created with Third Party Tools” on page 184.

Debugging NLMs Created with Third Party Tools

You can debug an NLM created with third party tools as long as the NLM meets these criteria:

- The NLM must use CodeView 5 (NB11) format.
- The NLM’s CodeView symbolic debug information must be in a separate file.
  This file must be named nlmName.NCV. For example, my_nlm.NCV.
- Each function must set up a standard x86 stack frame using the EBP register.

Failure to meet this rule causes these problems:
- The debugger’s call stack window will be inaccurate.
- The debugger’s Step Out command will not work.

NOTE Each called NLM you add to the Other Executables panel must include CodeView debugging information.

14. Click OK to save the settings.
15. Select Project > Make (F7) to build your project.

If you follow this procedure, each time you debug the NLM making the calls, the debugger loads each called NLM listed in the Other Executables panel.
This chapter defines the support libraries you must use to develop software using the CodeWarrior™ PDK for NetWare®.

This chapter contains these sections:

- Metrowerks Support Libraries
- Novell Support Libraries

### Metrowerks Support Libraries

The CodeWarrior Add-on Tools for NetWare includes a set of ANSI-compliant C++ support libraries. These libraries are called the Metrowerks Standard Libraries (MSL).

You must include some of these libraries in your NetWare projects.

**NOTE**

The CodeWarrior Add-on Tools for NetWare uses the standard C library included in the NetWare NDK.

For this reason, the NetWare PDK’s version of MSL does not include the standard C library.

### MSL Libraries Required by all Projects

For both C and C++ projects, you must add one of the libraries listed in Table 10.1 to your project.

These libraries contain startup and termination code that all programs require to run in the NetWare/x86 environment.
Table 10.1 MSL Libraries Required by C and C++ Projects

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mwcrtl.lib</td>
<td>Release version of the Metrowerks C runtime library. Add to your project’s release build target.</td>
</tr>
<tr>
<td>mwcrtld.lib</td>
<td>Debug version of the Metrowerks C runtime library. Add to your project’s debug build target.</td>
</tr>
</tbody>
</table>

These libraries are in the directory:

\{(Compiler)\Novell Support\Metrowerks Support\Libraries\Runtime\}

MSL Libraries for C++ Projects

If your project uses any of the classes defined in the MSL C++ class library, you must add one of the libraries listed in Table 10.2 to your project.

Table 10.2 MSL Libraries for C++ Development

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mwcpp.lib</td>
<td>Release version of the Metrowerks C++ class library. Add to your project’s release build target.</td>
</tr>
<tr>
<td>mwcppd.lib</td>
<td>Debug version of the Metrowerks C++ class library. Add to your project’s debug build target.</td>
</tr>
</tbody>
</table>

These libraries are in the directory:

\{(Compiler)\Novell Support\Metrowerks Support\Libraries\MSL C++\}

The CodeWarrior™ MSL C++ Reference manual documents the MSL C++ library.
Library Source Code and Library Project Files

The PDK includes all source code files used to build the MSL libraries. If you want, you can add these files to your projects instead of linking with the libraries.

The source code for the runtime library is in the directory:

\{Compiler\}\Novell Support\Metrowerks Support\Libraries\Runtime\(Sources\)

The source code for the standard C++ library is in the directory:

\{Compiler\}\Novell Support\Metrowerks Support\Libraries\MSL C++\Source

In addition, the PDK includes CodeWarrior projects that build the MSL libraries. Using these projects, you can modify a library’s source code and then rebuild the library.

Table 10.3 lists the MSL library projects.

<table>
<thead>
<tr>
<th>Project File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novell Runtime.mcp</td>
<td>Project file for the Novell/x86 version of the runtime library.</td>
</tr>
<tr>
<td>MSL C++.Novell.mcp</td>
<td>Project file for the Novell/x86 version of the MSL C++ library.</td>
</tr>
</tbody>
</table>

The file Novell Runtime.mcp is in the directory:

\{Compiler\}\Novell Support\Metrowerks Support\Libraries\Runtime

The file MSL C++.Novell.mcp is in the directory:

\{Compiler\}\Novell Support\Metrowerks Support\Libraries\MSL C++
Novell Support Libraries

The version of MSL included with the CodeWarrior Add-on Tools for NetWare does not include the standard C library.

As a result, if your project needs this library, your project must use the version included in Novell’s NetWare Development Kit (NDK).

For versions of the NDK older than the September 2001 version, the name of the NetWare standard C library is `clib.imp`.

This file is in the directory:
```
{NDK install dir}\Imports
```

NOTE

The NetWare NDK includes `clib.imp` for downward compatibility with projects created with older versions of the CodeWarrior Add-on Tools for NetWare.

To take advantage of improvements made to newer versions of the PDK, upgrade your old projects. To do this, follow the procedure in [Updating an Old Project](#).

For versions of the NDK dated September 2001 and later, the name of the NetWare standard C library is `libc.imp`. Use this library for all new projects.

This file is in the directory:
```
{NDK install dir}\libc\imports
```

In addition, if you use `libc.imp`, you must also add the file `libcpre.o` to your project.

This file is in the directory:
```
{NDK install dir}\libc\imports
```
Creating Static Libraries for NetWare/x86

This chapter defines what a static library is and then explains how to create a static library.

This chapter contains these sections:

- About Static Libraries
- Tutorial: Creating a Static Library

About Static Libraries

A static library is a collection of compiled object modules containing functions and data.

The linker copies the code and data your NLM references from the static library into your NLM. This is called static linking.

Your projects can use static libraries included with the CodeWarrior™ PDK for NetWare, third party static libraries, and static libraries you create yourself.

Tutorial: Creating a Static Library

To create a static library, follow these steps.

1. Run the CodeWarrior IDE.
2. Select File > New.

The IDE displays the New dialog box. This dialog box is shown in Figure 11.1.
3. Click the Project tab in the New dialog box.
   The New dialog box displays a list of wizards and project stationery in a list on its left and some text boxes on its right side.

4. Select NetWare Stationery.
   This option is in the list of wizards and project stationery on the left side of the New dialog box.

5. Enter the name of your new project in the Project name: text box.
   For this tutorial, enter the name `MyStaticLib`. 
6. Enter the directory in which to create your new project in the Location: text box.
   You can type this directory name directly into the Location: text box or you can press the Set... button.
   The Set... button displays the Create New Project... dialog box. You can use this dialog box to select the desired directory.
   For this tutorial, enter C:\NW_Projects\MyStaticLib.

7. Click OK.
   The IDE displays the New Project dialog box. This dialog box is shown in Figure 11.2.

Figure 11.2 The New Project Dialog Box

8. Select the project stationery named Static Library NLM.

   NOTE This stationery is in the Server\CLib group of project stationeries.
9. Click OK.

The CodeWarrior IDE creates a new project based on the Static Library NLM stationery.

The IDE names the project MyStaticLib.mcp and places it in the directory C:\NW_Projects\MyStaticLib.

After the IDE creates the project, it displays the project window shown in Figure 11.3.

**Figure 11.3 Project Window for MyStaticLib**

10. Add a function to the library.

   a. Select the file named staticlib.c

   b. Press Enter (or double-click filename)

      The IDE displays the contents of staticlib.c in an editor window.

      At this point, staticlib.c contains some comments and one simple function.

   c. Add the function shown in Listing 11.1 to staticlib.c.

      To do this, do one of these things:

      – Type the function’s code into the editor window.

      – Copy and paste the code from this document into the editor window.
Listing 11.1 Sample Code for the Static Library

```c
long multiply( int x, int y )
{
    return( x * y );
}
```

Figure 11.4 shows how the editor window should look after you enter the new function.

Figure 11.4 The Editor Window Containing the New Function

```c
/*
 * Sample Static Library for CodeWarrior
 * © 1998 Metrowerks Corp.
 * Questions and comments to:
 *   <mailto:support@metrowerks.com>
 *   <http://www.metrowerks.com>
*/

int foo( void )
{
    return 100;
}

long multiply( int x, int y )
{
    return( x * y );
}
```

d. Select File > Save

The CodeWarrior IDE saves the changes you made to staticlib.c.

e. Select File > Close

The editor window disappears.


The CodeWarrior IDE builds your static library.

The library’s name is staticlib.lib. It is in the directory C:\NW_Projects\MyStaticLib.
This completes the tutorial. Your static library is complete.

For instructions explaining how to use a static library, see Static Linking.
Targeting AMD Processors

This chapter discusses topics to consider when writing a NetWare® Loadable Module™ for the Advanced Micro Devices® AMD-K6® and Athlon™ family of processors.

This chapter contains these sections:

- **Overview: Support for AMD Processors**
- **Optimizations**
- **Pragmas**
- **Name Mangling**
- **Target Settings for AMD Processors**
- **Athlon-Specific Assembly Language Instructions**

**Overview: Support for AMD Processors**

The CodeWarrior™ NetWare/x86 C/C++ compiler can generate all instructions supported by the AMD-K6, AMD-K6-2, AMD-K6-III, and Athlon processors.

The compiler also fully supports AMD’s 3DNow™ technology.

For instructions that explain how to develop software for AMD processors, go to:

```
http://www.amd.com/us-en/Processors/
DevelopWithAMD/0,,30_2252,00.html
```

For online technical documentation of AMD processors, go to:

```
http://www.amd.com/us-en/Processors/
DevelopWithAMD/0,,30_2252,00.html
```
Optimizations

This section discusses coding techniques and compiler options you can use to create programs that run fast on AMD processors.

The topics in this section are:
- The MMX/3DNow! Calling Convention
- Minimizing Mode Switching
- Pragmas

The MMX/3DNow! Calling Convention

You can speed execution of MMX and 3DNow! instructions by enabling the Use MMX/3DNow! Calling Convention option. This option is in the x86 CodeGen target settings panel.

If you do this, the compiler generates code that uses an AMD processor’s registers more efficiently.

See x86 CodeGen Panel for an explanation of each item in the panel.

NOTE

If you decide to use the MMX/3DNow! calling convention, try to use it throughout your project. In other words, do not override the MMX/3DNow! calling convention, unless you have to.

The reason for this is that calling a function that does not use the MMX/3DNow! calling convention from a function that does, actually hinders performance.

Minimizing Mode Switching

To achieve fast execution on AMD processors, write code that reduces the number of floating point unit (FPU) mode switches. This is because a mode switch is an expensive operation.

A mode switch is a switch from the FPU’s x87 state to its MMX state and vice versa.

Mode switches are caused by mixing float and double code and by mixing 3DNow! code and non-3DNow! code.
One way to avoid mode switches is to use only `int` and `float` data types in programs that use 3DNow! extensions. If your program mixes `float` and `double` types in its 3DNow! code, an AMD processor must switch modes more often.

**TIP**

The default behavior of the NetWare/x86 C/C++ compiler makes a floating point macro a `double`.

To instruct the compiler to make a floating point macro a `float`, declare the constant as shown in **Listing 12.1**.

**Listing 12.1  Declaring a Constant that is a `float`**

```c
#define CONST_1 2.5f // CONST_1 macro is a float
#define CONST_2 2.5 //CONST_2 macro is a double
```

**Pragmas**

This section discusses the pragma statements you can use when writing code for AMD processors.

To learn more about `#pragma` statements, refer to the *CodeWarrior™ C Compilers Reference* manual.

The topics in this section are:

- [Code Generation Pragmas](#)
- [Constant Definition Pragmas](#)

**Code Generation Pragmas**

There are three pragmas you can use to enable the compiler’s MMX and 3DNow! code generation options.

These pragmas are:

- `#pragma mmx on | off | reset`
  
  Corresponds to the MMX option in the x86 CodeGen target settings panel.
Targeting AMD Processors
Pragmas

- `#pragma k63d on | off | reset`
  Corresponds to the 3DNow! option in the x86 CodeGen target settings panel.

**NOTE**
The `k63d` pragma is not compatible with Intel processors because Intel processors do not support 3DNow! multimedia extensions.

- `#pragma k63d_calls on | off | reset`
  Corresponds to the Use MMX/3DNow Calling Conventions option in the x86 CodeGen target settings panel.

**NOTE**
The `__cdecl`, `__stdcall`, and `__fastcall` declaration specifiers override the `#pragma k63d_calls` statement.

Refer to [x86 CodeGen Panel](#) for an explanation of each of these settings.

### Constant Definition Pragmas

The default behavior of the compiler is to make a floating-point macro a `double`. To define a floating-point macro that is a `float`, you must suffix the macro with an `f`. See [Listing 12.2](#).

**Listing 12.2** Default Floating-Point Constant Precision

```c
#define WIDTH 3.25; // compiler makes this constant a double
#define HEIGHT 4.5f; // compiler makes this constant a float
```

However, you can change the default behavior of the compiler with respect to floating-point constants by using the pragma `float_constants`.

**Listing 12.3** Example of the `#pragma float_constants` Pragma

```c
//syntax - #pragma float_constants on|off|reset

#pragma float_constants on     // constants are float by default
```

TNW–198  
(Targeting the NetWare® Operating System, PDK 5.0)
#pragma float_constants off // constants are double by default
#pragma float_constants on  // constants are float again
#pragma float_constants reset // constants are double again

## Name Mangling

The compiler mangles C++ names in code that uses the MMX/3DNow! calling convention differently from the way it mangles C++ names in code that does not use this calling convention.

This feature prevents you from accidentally linking 3DNow! code with non-3DNow! code.

**NOTE** The compiler uses the alternate name mangling scheme only when the MMX/3DNow! calling convention is enabled.

See [x86 CodeGen Panel](#) for an explanation of how to enable the MMX/3DNow! calling convention.
Targeting AMD Processors

Target Settings for AMD Processors

To configure a project to generate code for AMD processors:

1. Select Edit > targetname Settings

   The Target Settings window appears, as shown in Figure 12.1.

   **NOTE** The word targetname is a placeholder for the name of your project’s currently selected build target.

   **Figure 12.1** The x86 CodeGen Panel

2. Select the x86 CodeGen panel.

3. Select an AMD processor from the Target Processor list.

   The AMD processors in this list are:
   - AMD K6

TNW–200  Targeting the NetWare® Operating System, PDK 5.0
4. If you want to use MMX multimedia extensions, enable the Processor-specific instructions checkbox and the MMX option.

5. If you want to use 3DNow! multimedia extensions, enable the 3DNow! option.

6. If you want to use MMX/3DNow! calling conventions, enable the Use Multimedia Calling Convention option.

Refer to The MMX/3DNow! Calling Convention for an explanation of the purpose of this calling convention.

7. Click OK.

The IDE applies the selected options to your project’s current build target.

If you enable any of the options mentioned above, the compiler generates the AMD-specific instructions your program needs.

Athlon-Specific Assembly Language Instructions

AMD Athlon processors support MMX and 3DNow! instructions not supported by other processors.

The NetWare/x86 C/C++ compiler allows you to use these instructions in inline assembly language statements.

Refer to 80x86 Assembly Language Support for an explanation of the compiler’s support for inline assembly language.

This section contains these topics:

- Athlon-Specific 3DNow! Instructions
- Athlon-Specific MMX Instructions

Athlon-Specific 3DNow! Instructions

These 3DNow! instructions are specific to AMD’s Athlon processor.

- PF2IW
- PFNACC
Targeting AMD Processors
Athlon-Specific Assembly Language Instructions

- PFPNACC
- PI2FW
- PSWAPD

See your Athlon processor documentation for a definition of these instructions.

Athlon-Specific MMX Instructions

These MMX instructions are specific to AMD’s Athlon processor.

- MASKMOVQ
- MOVNTQ
- PAVGB
- PAVGW
- PEXTRW
- PMAXSW
- PINSRW
- PMAXUB
- PMINSW
- PMINUB
- PMOVMSKB
- PMULHUW
- PSADDBW
- PSHUFW
- PREFETCHNTA
- PREFETCH0
- PREFETCH1
- PREFETCH2
- SFENCE

See your Athlon processor documentation for a definition of these instructions.
NetWare Linker Command Files

This appendix defines the format of a CodeWarrior™ NetWare®/86 linker command file and explains each command that you can include in a linker command file.

This appendix contains these sections:
- Command File Format
- Command Definitions

Command File Format

A linker command file is a text file. The name of a linker command file must end with the suffix .DEF. Any word whose first character falls in the first column of the file is treated as a command.

A list of parameters can follow a command. Use commas or whitespace characters (spaces, tabs, and carriage returns) to separate each parameter.

A parameter name cannot start in the first column of a linker command file.

A hash character ( # ) starts a comment.

An at character ( @ ) followed by the name of another linker command file instructs the linker to process the second file by including its contents in the first file.
Command Definitions

The NetWare/x86 linker supports these linker command file commands:

- **AUTOUNLOAD**
- **CHECK**
- **COPYRIGHT**
- **DATE**
- **DEBUG**
- **DESCRIPTION**
- **EXIT**
- **EXPORT**
- **FLAG ON**
- **FLAG OFF**
- **HELP**
- **IMPORT**
- **LINKORDER**
- **MESSAGES**
- **MODULE**
- **MULTIPLE**
- **OS_DOMAIN**
- **PSEUDOPREEMPTION**
- **REENTRANT**
- **SCREENNAME**
- **STACK**
- **STACKSIZE**
- **START**
- **SYNCHRONIZE**
- **THREADNAME**
- **TYPE**
- **VERSION**
- **XDCDATA**

**NOTE** The CodeWarrior NetWare/x86 linker does not recognize these linker command file commands: CUSTOM, HIDESYM, INPUT, MAP, FULLMAP, PATH, SHARELIB, STAMPEDDATA, VERBOSE, OUT.
AUTOUNLOAD

Indicates that NetWare should unload the NLM when none of its previously used entry points are in use any longer.

Prototype  AUTOUNLOAD

See Also  FLAG ON, FLAG OFF, Table A.1

CHECK

Specifies the name of a function in the NLM that NetWare executes when the console operator tries to unload the NLM using the UNLOAD NetWare console command.

Prototype  CHECK  checkProcedureName

checkProcedureName

The name of the function for NetWare to call when the operator tries to unload the NLM using the UNLOAD console command.

Remarks  This function must return 0 if the NLM can be unloaded. It must return a non-zero value if the NLM cannot be unloaded.

See Also  START, EXIT

COPYRIGHT

Specifies the copyright text displayed when NetWare loads the NLM.

Prototype  COPYRIGHT "copyright string"

copyright string

The copyright text displayed when NetWare loads the NLM. This string must be bounded by double quotation marks ( "").
**Remarks**

If you do not use this command, your NLM displays no copyright information.

If you use this command, but include no string, your NLM displays Novell’s copyright notice as the default.

This command is equivalent to the Copyright option in the NLM Target panel. See **NLM Target Panel** for an explanation of this panel item.

**See Also**

**DESCRIPTION**

---

**DATE**

Time stamps the NLM.

**Prototype**

`DATE month, day, year`

- **month**
  
  The month to use in the time stamp.
  
  Must be between 1 and 12.

- **day**
  
  The day to use in the time stamp.
  
  Must be between 1 and 31.

- **year**
  
  The year to use in the time stamp.
  
  Must be a 4-digit year.
  
  Must be between 1900 and 3000.

**Remarks**

Separate the month, day, and year parameters with commas or whitespace (tabs, spaces, or carriage returns).

**See Also**

**VERSION**
NetWare Linker Command Files
Command Definitions

DEBUG

Includes information in the NLM needed by Novell’s debugger.

NOTE
You do not have to use this command to use the debugger included in the CodeWarrior™ PDK for NetWare®.

Prototype DEBUG

Remarks You must use this command to use the debugger included in the NetWare development kit. This program is called RDEBUG.

The DEBUG command is equivalent to the Generate Internal Debugger Records option in the NLM Target settings panel. See NLM Target Panel for an explanation of this panel item.

DESCRIPTION

Specifies the text displayed on the console screen when NetWare loads the NLM.

NOTE
Every NLM must have a description.

Prototype DESCRIPTION “descriptionString”

descriptionString
The text that NetWare displays on the console when it loads the NLM. This string must be bounded by double quotation marks.

Remarks Typically, the description contains the name of the NLM.

The description string can be up to 127 characters.

This command is equivalent to the Description item in the NLM Target settings panel. See NLM Target Panel for an explanation of this panel item.
EXIT

Specifies the name of a function in the NLM that performs cleanup.

**NOTE**

An NLM must have an exit function.

**Prototype**

EXIT exitProcedureName

exitProcedureName

Name of the function in the NLM that performs cleanup.

**Remarks**

Typically, the exit routine ensures that all resources have been released and all threads terminated before the NLM unloads.

If you do not use this option, the linker uses these default routines:

_Stop for projects that use clib.imp.

_LibCPostlude for projects that use libc.imp.

This command is equivalent to the Exit option in the NLM Linker panel. See [NLM Linker Panel](#) for an explanation of this panel item.

**See Also**

CHECK, START

EXPORT

Lists the symbolic name of each variable and function the NLM makes available to other NLMs.

**Prototype**

EXPORT symbolList

symbolList

List of symbols exported by the NLM.

**Remarks**

Each symbol must be separated by a comma or whitespace character (tab, space, or carriage return).
Each symbol that appears on a new line must be indented by whitespace.

Instead of listing each symbol after the `EXPORT` keyword, you can enter the name of a file that contains the symbol list. This file name must be preceded by the at character (@).

| NOTE | Instead of using the `EXPORT` command, you can add a `.EXP` file to your project. Refer to Symbol Export Files for more information. |

See Also: `IMPORT`

### FLAG_ON

Sets the specified bit flag in the NLM’s header.

Each bit flag controls a different feature.

**Prototype**

```plaintext
FLAG_ON flagNumber

flagNumber

The bit flag to turn on.

Must be one of the values listed in Table A.1.
```

**Remarks**

You can use `FLAG_ON` and `FLAG_OFF` more than once in a linker command file. The result is cumulative.

This command is equivalent to the Flags option in the NLM Linker settings panel. See NLM Linker Panel for an explanation of this panel item.

See Also: `FLAG_OFF`

### FLAG_OFF

Clears the specified bit flag in the NLM’s header.
Each bit flag controls a different feature.

Prototype

\[
\text{FLAG\_OFF flagNumber}
\]

flagNumber

The bit flag to turn off.

Must be one of the values listed in Table A.1.

Remarks

You can use \text{FLAG\_ON} and \text{FLAG\_OFF} more than once in a linker command file. The result is cumulative.

This command is equivalent to the Flags option in the NLM Linker settings panel. See NLM Linker Panel for an explanation of this panel item.

See Also

\text{FLAG\_ON}

\hline

\textbf{HELP}

Specifies the directory path for the file that contains the NLM's help screens.

Prototype

\[
\text{HELP helpFilePath}
\]

helpFilePath

Directory path for file containing NLM help screens.

Remarks

A help file name must end in the suffix \text{.HLP}.

\textbf{NOTE}

Instead of using the \text{HELP} command, you can add a \text{.HLP} file to your project. Refer to Help Screen Files for more information.

See Also

\text{MESSAGES}
IMPORT

Lists the symbolic name of each variable and function defined in other NLMs that this NLM uses.

Prototype

```
IMPORT symbolList
```

symbolList

List of symbols imported by the NLM.

Remarks

Each symbol must be separated by a comma or whitespace character (tab, space, or carriage return).

Each symbol that appears on a new line must be indented by whitespace.

Instead of listing each symbol after the IMPORT keyword, you can enter the name of a file that contains the symbol list. This file name must be preceded by an at character (@).

NOTE

Instead of using the IMPORT command, you can add a .IMP file to your project. Refer to Symbol Import Files for more information.

See Also

EXPORT

LINKORDER

Specifies the functions and variables that the linker should link first.

NOTE

Use this command to specify symbols you want to link first. You do not need to list all the symbols in your program.

Prototype

```
LINKORDER symbolList
```

symbolList

List of functions and variables that the linker should link first.
Each symbol must be separated by a comma or whitespace character (tab, space, or carriage return). Each symbol that appears on a new line must be indented by whitespace.

Instead of listing each symbol after the LINKORDER keyword, you can enter the name of a file that contains the symbol list. This file name must be preceded by an at character (@).

You can specify code (function) and data (variable) symbols in any order in the symbol list. However, the linker segregates the different symbol types into separate sections in the link file in this order:

1. Code items
2. Data items
3. BSS data items

For example, if you specify the following symbol list:

   CODE1
   CODE2
   DATA1
   CODE3
   DATA2

The order of these symbols in the file produced by the linker is:

   CODE1
   CODE2
   CODE3
   (other code items)
   DATA1
   DATA2
   (other data items)
   (BSS data items)

**NOTE** Instead of using the LINKORDER command, you can add a .ORD file to your project. Refer to [Link Order Files](#) for more information.
MESSAGES

Specifies path to directory that contains the NLM’s message file.

Prototype: MESSAGES messageFilePath

messageFilePath

The directory path for the file containing the NLM’s messages.

Remarks: The name of the file in the messageFilePath directory must end with the suffix .MSG.

NOTE: Instead of using the MESSAGES command, you can add a .MSG file to your project. Refer to Message Files for more information.

MODULE

Specifies the list of NLMs that NetWare must load before loading this NLM.

 Prototype: MODULE autoloadNLMList

autoloadNLMList

List of NLMs that NetWare must load before loading this NLM.

Remarks: The autoload list should include the NLMs that export items that this NLM imports.

Each NLM name must be separated by a comma or whitespace character (tab, space, or carriage return).

Each NLM name that appears on a new line must be indented by whitespace.

Instead of listing each symbol after the MODULE keyword, you can enter the name of a file that contains the NLM list. This file name must be preceded by an at character (@).
**MULTIPLE**

Indicates that NetWare can load multiple copies of the NLM.

Prototype  MULTIPLE

Remarks  If you do not use this command, NetWare can load just one copy of the NLM.

See Also  FLAG_ON, FLAG_OFF, Table A.1

**OS_DOMAIN**

Indicates that the NLM must be loaded in the operating system’s domain.

Prototype  OS_DOMAIN

See Also  FLAG_ON, FLAG_OFF, Table A.1

**PSEUDOPREEMPTION**

Indicates that NetWare can force the NLM to relinquish control of the processor if the NLM does not do so often enough on its own.

Prototype  PSEUDOPREEMPTION

Remarks  Use the NetWare console command Set Pseudo Preemption Time to set the amount of time that passes before NetWare forces the NLM to relinquish control.

When the NLM exceeds this time limit, NetWare regains control at the next file read or write system call the NLM makes.

See Also  FLAG_ON, FLAG_OFF, Table A.1
### REENTRANT

Indicates that the NLM’s code can be executed by multiple threads at the same time.

**Prototype**

REENTRANT

**Remarks**

NetWare loads one copy of a reentrant NLM.

**See Also**

Table A.1

### SCREENNAME

Specifies the name of the screen the NLM displays when it is loaded.

**Prototype**

SCREENNAME “initialScreenName”

*initialScreenName*

The name of the screen the NLM displays when it is loaded.

This name must be 71 characters or less and must be bound by double quotation marks.

**See Also**

DESCRIPTION

### STACK

Defines the size of the NLM’s stack (in bytes).

**Prototype**

STACK stackSize

*stackSize*

The size of the NLM’s stack (in bytes).

**Remarks**

The minimum stack size is 2 KB (2048). Over 4 KB (4096) is recommended. The default stack size is 8 KB (8192).
NetWare Linker Command Files
Command Definitions

This command is equivalent to the Stack Size option in the NLM Target panel. See NLM Target Panel for an explanation of this panel item.

This command is equivalent to the STACKSIZE linker command file command.

See Also STACKSIZE

STACKSIZE

Defines the size of the NLM’s stack (in bytes).

Prototype

STACKSIZE stackSize

stackSize

The size of the NLM’s stack (in bytes).

Remarks

The minimum stack size is 2 KB (2048). Over 4 KB (4096) is recommended. The default stack size is 8 KB (8192).

This command is equivalent to Stack Size option in NLM Target panel. See NLM Target Panel for an explanation of this panel item. This command is equivalent to the STACK linker command.

See Also STACK

START

Specifies the name of a function in the NLM where execution starts.

NOTE

Every NLM must have a start procedure.

Prototype

START startProcedureName

startProcedureName

Name of function in NLM where execution starts.
Remarks

If you do not use this command, the linker uses these default start procedures:

- _Prelude for projects that use clib.imp.
- _LibCPrelude for projects that use libc.imp.

This command is equivalent to the Start option in the NLM Linker panel. See NLM Linker Panel for an explanation of this panel item.

See Also
CHECK, EXIT

SYNCHRONIZE

Indicates that NetWare should stop processing console commands while the NLM is loading.

Prototype
SYNCHRONIZE

See Also
FLAG_ON, FLAG_OFF, Table A.1

THREADNAME

Specifies the prefix for NetWare to use to name the NLM threads.

Prototype
THREADNAME initialThreadName

initialThreadName

The prefix that NetWare uses to name the NLM’s threads.

Remarks

The first 12 characters of initialThreadName are used to create names for the NLM’s threads. For example, if initialThreadName is MyProcess, then the threads created in the NLM are named MyProcess 1, MyProcess 2, and so on.

You can display thread names in the debugger.

See Also
SCREENNAME
TYPE

Specifies the kind of service the NLM provides.

Prototype

```
TYPE typeNumber
```

typeNumber

The type of the NLM.

Table A.2 lists and defines the values you can specify for typeNumber.

Remarks

This command can appear just once in a linker command file.

This command is equivalent to the Output Type option in the NLM Target panel. See NLM Target Panel for an explanation of this panel item.

VERSION

Specifies the version of the NLM.

NOTE

An NLM must have a version number.

Prototype

```
VERSION majorVersion, minorVersion, [ revision ]
```

majorVersion

The first number in the NLM’s version number.

This value of this parameter can be anything.

minorVersion

The second number in the NLM’s version number.

This value must be from 0 to 99.

revision

The value specified for this parameter must be from 0 to 26.
If it is 0, no revision character appears when the NLM loads. If it is 1 through 26, the letter of the alphabet appears, which corresponds to the value specified.

This parameter is optional.

**Remarks**
The version number appears on the console screen when the NLM loads.

An example version number is `1.3.b`.

The `majorVersion`, `minorVersion`, and `revision` parameters must be separated by whitespace characters (spaces, tabs, or carriage returns).

Parameters that appear on a new line must be indented with at least one space or tab.

This command is equivalent to the Major, Minor, and Revision options in the NLM Target panel. See [NLM Target Panel](#) for an explanation of these panel items.

**See Also**  
[DATE](#)

---

**XDCDATA**

Specifies path to a file containing Remote Procedure Call (RPC) descriptions.

**Prototype**

```
XDCDATA rpcFilePath
```

- `rpcFilePath`  
  Directory path for a file containing RPC descriptions.

**Remarks**
The name of a Remote Procedure call file must end in the suffix `.RPC`.

**NOTE**  
Instead of using the `XDCDATA` command, you can add a `.RPC` file to your project. Refer to [Remote Procedure Call Files](#) for more information.
NOTE The RPC compiler (which produces RPC files) has not yet been released.

See Also EXPORT

### Table A.1  FLAGS_ON/FLAGS_OFF Commands: Arguments

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Defines whether an NLM is reentrant. If this flag is set, more than one thread can execute the code of an NLM at the same time. This flag is equivalent to the <strong>REENTRANT</strong> linker command.</td>
</tr>
<tr>
<td>2</td>
<td>Defines how many copies of an NLM can be loaded. If this flag is set, NetWare can load more than one copy of the NLM. This flag is equivalent to the <strong>MULTIPLE</strong> linker command.</td>
</tr>
<tr>
<td>4</td>
<td>Controls how NetWare loads an NLM. If this flag is set, NetWare’s load process sleeps until the NLM calls <code>SynchronizeStart()</code>(). This mechanism prevents NetWare from processing other console commands while the NLM is loading. This flag is equivalent to the <strong>SYNCHRONIZE</strong> linker command.</td>
</tr>
<tr>
<td>8</td>
<td>Controls whether an NLM can be preempted. If this flag is set, NetWare forces the NLM to relinquish control of the processor if the NLM does not do so often enough. This flag is equivalent to the <strong>PESEUDOPREEMPTION</strong> linker command.</td>
</tr>
<tr>
<td>16</td>
<td>Controls whether an NLM is loaded in the domain of the operating system. If this flag is set, the NLM is loaded into the operating system’s domain. This flag is equivalent to the <strong>OS_DOMAIN</strong> linker command.</td>
</tr>
<tr>
<td>64</td>
<td>Controls when an NLM is unloaded. If this flag is set, NetWare unloads the NLM when none of its entry points are in use. This flag is equivalent to the <strong>AUTOUNLOAD</strong> linker command.</td>
</tr>
</tbody>
</table>
### Table A.2  NLM Types and File Name Extension

<table>
<thead>
<tr>
<th>Type Number</th>
<th>Extension</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>.NLM</td>
<td>Generic NLM (default)</td>
</tr>
<tr>
<td>1</td>
<td>.LAN</td>
<td>LAN driver</td>
</tr>
<tr>
<td>2</td>
<td>.DSK</td>
<td>Disk driver</td>
</tr>
<tr>
<td>3</td>
<td>.NAM</td>
<td>Name space support module</td>
</tr>
<tr>
<td>4</td>
<td>.NLM</td>
<td>Utility or support program</td>
</tr>
<tr>
<td>5</td>
<td>.MSL</td>
<td>Mirrored server link</td>
</tr>
<tr>
<td>6</td>
<td>.NLM</td>
<td>Operating system NLM</td>
</tr>
<tr>
<td>7</td>
<td>.NLM</td>
<td>Paged high operating system NLM</td>
</tr>
<tr>
<td>8</td>
<td>.HAM</td>
<td>Host adapter module (works with custom device module)</td>
</tr>
<tr>
<td>9</td>
<td>.CDM</td>
<td>Custom device module (works with host adapter module)</td>
</tr>
<tr>
<td>10</td>
<td>.NLM</td>
<td>Reserved</td>
</tr>
<tr>
<td>11</td>
<td>.NLM</td>
<td>Reserved</td>
</tr>
<tr>
<td>12</td>
<td>.NLM</td>
<td>Reserved</td>
</tr>
</tbody>
</table>
Index

Symbols
#pragma calling_convention 76
#pragma float_constants 198
#pragma use_frame 82
_asm keyword 93
_builtin_ 79
_cdecl 73, 198
_declspec (naked) 95
fastcall 75, 198
stdcall 74, 198

A
allocating variables 77
AMD Athlon-specific instructions 201–202
AMD processors
    Athlon-specific instructions 201–202
    name mangling 199
    optimizations 196–197
        minimizing mode switching 196–197
        MMX/3DNow! calling convention 196
    overview of support for 195
    pragmas 197–199
        code generation 197–198
            #pragma k63d 198
            #pragma k63d_calls 198
            #pragma mmx 197
        constant definition 197–199
            #pragma float_constants 198
        related documentation 195
        target settings 200–201
ANSI Keywords Only target setting 92, 93
ANSI Strict target setting 92, 93
asm keyword 92
assembler, command line 26
assembly language, inline 89, 91–105
    asm keyword 93
    _declspec (naked) 95
AMD processors 104
ANSI Keywords Only option 92
ANSI Strict option 92
asm keyword 92–93
blocks within functions 95–96
comments 100
complete functions 95
directives 97–99
    _emit 98
    align 98
db 98
dd 98
dw 98
    nooptimize 99, 104
    optimize 99, 104
directives, table of 98
inlining functions 103
local variables and MMX registers 102–103
    non-MMX context, defined 102
    optimization 103–104
preprocessor support 100
statement labels 99
support for Microsoft extensions 93
syntax rules 93–94
valid operands 101
within macros 96–97
AUTOUNLOAD linker command 205

B
build target, defined 22

C
C/C++ compiler, see compiler
calling conventions 71–77
C language 72–73
C++ language 74
    fast call 74–75
    overriding default 75–77
Pascal 74
CHECK linker command 205
CodeWarrior development process 28
COFF, see Common Object File Format
command line compiler 149
command line linker 158
command line tools
    compiler 149
        C/C++ language options 133–138
            -ansi 133
            -ARM 134
            -bool 134
            -char 134
## Index

- `C/C++` warning options 145–148
  - `warnings` 145
- Code optimization options 142–145
  - `-O` 142
  - `-O+` 142
  - `-opt` 143
- Debugging control options 148, 149
  - `-g` 148
  - `-sym` 148
- Preprocessing options 128–133
  - `-c` 128
  - `-codegen` 128
  - `-convertpaths` 128
  - `-cwd` 128
  - `-defaults` 129
  - `-define` 129
  - `-disassemble` 129
  - `-E` 129
  - `-EP` 129
  - `-ext` 130
  - `-gccincludes` 130
  - `-I-` 130
  - `-I+` 130
  - `-include` 130
  - `-ir` 131
  - `-keepobjects` 131
  - `-M` 131
  - `-make` 131
  - `-MD` 131
  - `-MM` 131
  - `-MMD` 131
  - `-nofail` 131
  - `-nolink` 131
  - `-noprecompile` 132
  - `-nosyspath` 132
  - `-o` 132
  - `-P` 132
  - `-precompile` 132
  - `-prefix` 133
  - `-preprocess` 133
  - `-S` 133
  - `-stdinc` 133
  - `-undefined` 133
- x86 code generation options 139
  - `-abi` 139
  - `-align` 139
  - `-exceptions` 139
  - `-instructionset` 140
  - `-mangling` 140
  - `-processor` 141
- Global options 127
  - `-help` 125
  - `-maxerrors` 127
  - `-maxwarnings` 127
  - `-msgstyle` 127
  - `-search` 127
  - `-stderr` 127
  - `-stdinc` 127
  - `-stdkeyword` 127
  - `-verbose` 126
  - `-version` 126
  - `-wraplines` 127
- Linker 158
  - Debugging control options 154
    - `-g` 154
    - `-osym` 154
    - `-sym` 154
  - Disassembler options 155–158
    - `show` 155
  - General options 149, 150
    - `-defaults` 150
    - `-disassemble` 149
    - `-L+` 149
    - `-I+` 150
    - `-Lr` 150
    - `-nofail` 150
    - `-S` 150
    - `-stdlib` 150
  - NLMversion 153
  - Screenname 153
  - Warning options 155
    - `-warnings` 155
Index

x86 COFF options 152
  -filealign 152
  -flags 152
x86 linker options 151–152
  -addcommandline 151
  -checksum 151
  -defaultlibs 151
  -export 151
  -f 151
  -map 152
  -mergesects 152
  -noentry 152
  -zerobss 152
command line tools, see development tools
Common Object File Format 108
compiler
  additional documentation 69
AMD processors
    Athlon-specific instructions 201–202
    name mangling 199
    target settings 200–201
calling conventions 71–77
  C language 72–73
  C++ language 74
  fast call 74–75
  overriding default 75–77
  Pascal 74
data type formats 70–71
floating-point data types 71
  double 71
  float 71
  long double 71
  short double 71
inline assembly language 89, 91–105
  __asm keyword 93
  __declspec(naked) 95
AMD processors 104
ANSI Keywords Only option 92
ANSI Strict option 92
asm keyword 92–93
blocks within functions 95–96
comments 100
complete functions 95
directives 97–99
  _emit 98
  align 98
  db 98
  dd 98
  dw 98
  nooptimize 99, 104
  optimize 99, 104
directives, table of 98
inlining functions 103
local variables and MMX registers 102–103
  non-MMX context, defined 102
optimization 103–104
preprocessor support 100
statement labels 99
support for Microsoft extensions 93
syntax rules 93–94
valid operands 101
within macros 96–97
integer data types 70–71
  bool 70
  char 70
  int 71
  long 71
  long long 71
  short 70
  signed char 70
  unsigned char 70
  unsigned int 71
  unsigned long 71
  unsigned short 71
intrinsic functions
  __builtin__ prefix 79–80
  defined 78
  inlining 77–80
  x86 port intrinsic functions 80
optimizations 82–88
  for AMD processors 196–197
    minimizing mode switching 196–197
  instruction scheduling 84
  loop distribution 87
  loop vectorization 84–87
    dataflow restrictions 86–87
    loop restrictions 86
  reduction functions 88
  register coloring 82–84
  register variables 81
  vectorization, defined 84
pragmas 89
  #pragma calling_convention 76
  #pragma float_constants 198
  #pragma use_frame 82
  for AMD processors 197–199
    code generation 197–198
    #pragma k63d 198
Targeting the NetWare® Operating System, PDK 5.0

Index

#pragma k63d_calls 198
#pragma mmx 197
constant definition 198-199
#pragma float_contents 198

See also C Compilers Reference

register keyword 81
variable allocation 77
See also C Compilers Reference
See also IDE User Guide

COPYRIGHT linker command 205

D

DATE linker command 206
deadstripping unused code 121
defined 121
DEBUG linker command 207
debugger
project settings
  CodeView format 164-166
  x86 processor settings panel 166
  runtime settings panel 166-168
setup procedure 159-163
  Ethernet-TCP/IP link 162-163
  global connection panel 161, 163
  serial link 160-162
shared library NLM, debugging 177
  add the library to the project 178
  use other executables settings panel 179
static library, debugging 184
third party NLMs, requirements for debugging 184
tutorial 168-176
  debugger log window 175
  debugger main window 175
  view NetWare server processes 176-177
debugging an NLM, tutorial 168-176
description linker command 207
development process, CodeWarrior 28
development tools 18-27
  command line tools 26
  compiler 24
debugger 25-26
editor 24
  integrated development environment 18
description linker command 207
Metrowerks Standard Libraries 26
project manager 22
  project, defined 22

PROJECT linker command, defined 22
directive, defined 97
documentation, see related documentation
dynamic linking 120

E
editor, see development tools
environment variables, see installation procedure
Ethernet-TCP/IP debugging setup procedure 162-163
EXIT linker command 208
EXPORT linker command 208

F
file search order, linker 122
FLAG_OFF linker command 209
  arguments 220
FLAG_ON linker command 209
  arguments 220
floating-point data types 71
double 71
float 71
long double 71
short double 71
format, linker command file 203

H
HELP linker command 210
help screen file 112
host, defined 22

I
IMPORT linker command 211
inline assembly language 89, 91-105
__asm keyword 93
__declspec(naked) 95
AMD processors 104
ANSI Keywords Only option 92
ANSI Strict option 92
asm keyword 92-93
blocks within functions 95-96
comments 100
code complete functions 95
directives 97-99
.emit 98
align 98
Index

db 98
dd 98
dw 98
nooptimize 99, 104
optimize 99, 104
directives, table of 98
inlining functions 103
local variables and MMX registers 102–103
    non-MMX context, defined 102
optimization 103–104
preprocessor support 100
statement labels 99
support for Microsoft extensions 93
syntax rules 93–94
valid operands 101
within macros 96–97

installation procedure 15–18
NovellNDK environment variable 16

integer data types 70–71
    bool 70
    char 70
    int 71
    long 71
    long long 71
    short 70
    signed char 70
    unsigned char 70
    unsigned int 71
    unsigned long 71
    unsigned short 71

integrated development environment, see development tools
intrinsic functions
    __builtin__ prefix 79–80
    defined 78
    inlining 77–80
    x86 port intrinsic functions 80

L

libraries
    shared NLM 120
    static 120
    support
        Metrowerks 185–187
        CodeWarrior projects for 187
        for C++ projects 186
        required 185–186
        source code 187

Novell 188
link order 121–122
link order file 112
link order view 121
linker 107–122
deadstripping 121
    defined 121
dynamic linking 120
file search order 122
input files 107–112
command files 109, 203–221
commands 204–221
    AUTOUNLOAD 205
    CHECK 205
    COPYRIGHT 205
    DATE 206
    DEBUG 207
    DESCRIPTION 207
    EXIT 208
    EXPORT 208
    FLAG_OFF 209
        arguments 220
    FLAG_ON 209
        arguments 220
    HELP 210
    IMPORT 211
    LINKORDER 211
    MESSAGES 213
    MODULE 213
    MULTIPLE 214
    OS_DOMAIN 214
    PSEUDOPREEMPTION 214
    REENTRANT 215
    SCREENNAME 215
    STACK 215
    STACKSIZE 216
    START 216
    SYNCHRONIZE 217
    THREADNAME 217
    TYPE 218
        arguments 221
    VERSION 218
    XDCDATA 219
    format 203

Common Object File Format
    object modules 108
    static libraries 108
    help screen files 112
    link order files 112
Index

- message files 112
- NetWare Loadable Modules 108
- object modules 108
  - acceptable formats 108
  - defined 108
- remote procedure call files 112
- static libraries 108
- symbol export files 109-110
  - symbol prefixes 110
- symbol import files 110-111
  - symbol prefixes 111
- libraries, shared NLM 120
- libraries, static 120
- link order 121-122
- link order view 121
- map file
  - defined 113
  - example 114
- output files
  - map file 113-119
  - NetWare Loadable Module 113
    - types
      - custom device 113
      - disk driver 113
      - generic 113
      - host adapter 113
      - LAN driver 113
      - mirrored server link 113
      - name space support 113
      - operating system 113
      - paged high operating system 113
      - utility 113
- static linking 120
- linker command file 109, 203-221
- commands 204-221
  - AUTOUNLOAD 205
  - CHECK 205
  - COPYRIGHT 205
  - DATE 206
  - DEBUG 207
  - DESCRIPTION 207
  - EXIT 208
  - EXPORT 208
  - FLAG_OFF 209
    - arguments 220
  - FLAG_ON 209
    - arguments 220
  - HELP 210
  - IMPORT 211
  - LINKORDER 211
  - MESSAGES 213
  - MODULE 213
  - MULTIPLE 214
  - OS_DOMAIN 214
  - PSEUDOPREEMPTION 214
  - REENTRANT 215
  - SCREENNAME 215
  - STACK 215
  - STACKSIZE 216
  - START 216
  - SYNCHRONIZE 217
  - THREADNAME 217
  - TYPE 218
    - arguments 221
  - VERSION 218
  - XDCDATA 219
- format 203
- linker command file format 203
- linker input files 107-112
- LINKORDER linker command 211

M

- map file 113-119
  - defined 113
  - example 114
- message file 112
- MESSAGES linker command 213
- Metrowerks Standard Libraries 26, 27, 185, 185-187
- MODULE linker command 213
- Mpkxdc panel 54
- MSL, see Metrowerks Standard Libraries
- MULTIPLE linker command 214

N

- name mangling for AMD processors 199
- NetWare development tools, see development tools
- NetWare Loadable Module 113
  - defined 113
    - types
      - custom device 113
      - disk driver 113
      - generic 113
      - host adapter 113
      - LAN driver 113
      - mirrored server link 113
      - name space support 113
Index

object library 113
operating system 113
page high operating system 113
utility 113
NetWare server processes, view 176–177
NLM, see NetWare Loadable Module
Novell NDK environment variable

O
object module 108
acceptable formats 108
defined 108
optimizations 82–88
for AMD processors 196–197
minimizing mode switching 196–197
MMX/3DNow! calling convention 196
instruction scheduling 84
loop distribution 87
loop vectorization 84–87
dataflow restrictions 86–87
loop restrictions 86
reduction functions 88
register coloring 82–84
register variables 81
vectorization, defined 84
OS_DOMAIN linker command 214

P
platform target, defined 22
pragmas

#pragma calling_convention 76
#pragma float_constants 198
#pragma use_frame 82
for AMD processors 197–199
code generation 197–198
#pragma k63d 198
#pragma k63d_calls 198
#pragma mmx 197
constant definition 198–199
#pragma float_constants 198
See also C Compilers Reference
preference panels
global connection 63, 66
project
build target, defined 22
build target, example 23
creating, tutorial 31
defined 22
host, defined 22
platform target, defined 22
preference panels
global connection 63, 66
related terms 22
required files 67
target settings
general 39
NetWare-specific 39, 66
overview 37
panels
access paths 47
NLM linker 57, 60
NLM target 47–50
runtime settings 43, 45
source trees 45
target settings 40
x86 disassembler 60, 63
x86 processor 50
updating old 34, 36
project manager, see development tools
project stationery
defined 29
NetWare stationery 30, 31
purpose 29
PSEUDOPREEMPTION linker command 214

R
r 138
REENTRANT linker command 215
register keyword 81
related documentation
Metrowerks 10
Novell 10
release notes 11
remote procedure call file 112
root paths, defined 45

S
SCREENNAME linker command 215
serial debugging setup procedure 160–162
shared library NLM
debugging 177
add the library to the project 178
use other executables settings panel 179
defined 120
software installation, see installation procedure
Index

STACK linker command 215
STACKSIZE linker command 216
START linker command 216
static 189
static library
  debugging 184
  defined 120, 189
  linker input file 108
  tutorial 189–194
static library, defined 189
static linking 120
stationery, see project stationery
support libraries
  Metrowerks 185–187
  CodeWarrior projects for 187
  for C++ projects 186
  required 185–186
  source code 187
  Novell 188
symbol export file 109–110
  defined 109
  symbol prefixes 110
symbol import file 110–111
  defined 110
  symbol prefixes 111
symbol prefixes 110, 111
SYNCHRONIZE linker command 217
system requirements 14–15
  NetWare server 14
  Windows PC 14

T

target settings
  general 39
  NetWare-specific 39, 66
  overview 37
  panels
    access paths 47
    NLM linker 57, 60
    NLM target 47–50
    runtime settings 43, 45
    source trees 45
    target settings 40
    x86 disassembler 60, 63
    x86 processor 50
target settings for AMD processors 200–201

third party NLMs, requirements for debugging 184
THREADNAME linker command 217
tutorial
  creating a new project 31
  creating a static library 189–194
  debugging an NLM 168–176
  debugger log window 175
  debugger main window 175
TYPE linker command 218
  arguments 221

U

updating an old project 34, 36

V

VERSION linker command 218
view NetWare server processes 176–177

X

XDCDATA linker command 219