

# CodeWarrior®

## Debugger User Guide



CodeWarrior は出荷直前でも改良されることがあるので、このマニュアルに記載されている内容の一部が本物のソフトウェアの動きと異なることがあるかもしれません。最新の情報については CodeWarrior の「Release Notes」フォルダをご覧ください。

Revised: 980831-mds-JP981029

Metrowerks CodeWarrior © Copyright 1993-1998 by Metrowerks Inc. and its Licensors. All rights reserved.

お客様は、本 CD に記録されている文書を個人使用目的に限り、プリントすることができます。この場合を除いて、Metrowerks Inc. からの書面による承諾なしに、本 CD に記録されている文書の全部、または、一部をいかなる形態、方法（電子的、物理的な複製、または、写真複写、録音録画、その他すべての情報記録、再生システムを含む）により、複製または伝達することを禁じます。

Metrowerks の名称、ロゴ、CodeWarrior、Software at Work は、Metrowerks Inc. の登録商標です。

PowerPlant、PowerPlant Constructor は、Metrowerks Inc. の商標です。

記載の商標および登録商標は、各社が保有します。

CD に記録されているすべてのソフトウェアおよび文書は、CodeWarrior QuickStart の巻末に記述されているライセンス契約が適用されます。

連絡先：

---

Japan	メトロワークス株式会社 150-0042 東京都渋谷区宇田川町 36-6 ワールド宇田川ビル 8F TEL : (03) 3780-6091 FAX : (03) 3780-6092
U.S.A.	Metrowerks Corporation 9801 Metric Boulevard, Suite 100 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
WWW サーバ	<a href="http://www.metrowerks.com">http://www.metrowerks.com</a>
ユーザー登録	<a href="mailto:j-register@metrowerks.com">j-register@metrowerks.com</a>
テクニカルサポート	<a href="mailto:j-support@metrowerks.com">j-support@metrowerks.com</a>
購入 / 契約更新	<a href="mailto:j-sales@metrowerks.com">j-sales@metrowerks.com</a>
インフォメーション	<a href="mailto:j-info@metrowerks.com">j-info@metrowerks.com</a>

---

# 目次

---

第 1 章 紹介	7
このマニュアルの概要	7
Metrowerks の西暦 2000 年問題対応	8
リリースノートについて	8
マニュアルの表記規則	8
表記について	8
ホストについて	9
画面図について	9
キーボードについて	9
新機能	10
システムの必要条件	11
Windows	11
Mac OS.	11
Solaris	11
MW Debug をインストール	12
出発点	13
さらに学ぶために	13

---

第 2 章 はじめに	15
この章の概要	15
デバッグの準備	15
デバッグのためにターゲットを準備	15
デバッグのためにファイルを設定	16
シンボル情報を生成	17
デバッグを起動	18
統合デバッグを使う	18
MW Debug を IDE から起動 ( Mac OS )	18
MW Debug を直接起動	19
シンボルファイル	19

---

第 3 章 デバッグのウィンドウ	21
ウィンドウの概要	21
プログラムウィンドウ	21
スタッククロール欄	22
変数欄	23
デバッグのツールバー	24
ソース欄	25
ブラウザウィンドウ	29
ファイル欄	31
関数欄	32

大域変数欄 . . . . .	33
ブラウザのソース欄 . . . . .	33
関数ポップアップメニュー . . . . .	34
Expression ウィンドウ . . . . .	35
Breakpoint ウィンドウ . . . . .	35
Watchpoint ウィンドウ . . . . .	36
Log ウィンドウ . . . . .	37
変数ウィンドウ . . . . .	37
配列ウィンドウ . . . . .	38
メモリウィンドウ . . . . .	39
レジスタウィンドウ . . . . .	41
プロセスウィンドウ . . . . .	43
プロセス欄 . . . . .	44
タスク欄 . . . . .	45
プロセスウィンドウのツールバー . . . . .	45
 第 4 章 デバッガの使い方 . . . . .	47
デバッガの概要 . . . . .	47
デバッガを起動する . . . . .	47
コードの実行、ステップ実行、停止 . . . . .	49
カレントステートメント矢印 . . . . .	51
コードを実行 . . . . .	51
一行ずつ実行する . . . . .	52
ルーチンの中へ入る . . . . .	53
ルーチンの中から呼び出し元に戻る . . . . .	53
ステートメントをスキップする . . . . .	54
実行を停止する . . . . .	55
プログラムを終了する . . . . .	56
ナビゲーション . . . . .	56
一般的なナビゲーション . . . . .	57
コールチェーンによるナビゲーション . . . . .	57
ブラウザウィンドウによるナビゲーション . . . . .	58
ソースコードによるナビゲーション . . . . .	60
Find ダイアログの使い方 . . . . .	62
フォントや色を変更する . . . . .	63
ブレークポイント . . . . .	63
ブレークポイントを設定 . . . . .	64
ブレークポイントを消去 . . . . .	64
一時的なブレークポイント . . . . .	65
ブレークポイントを表示 . . . . .	65
条件ブレークポイント . . . . .	66
ブレークポイントでコードを最適化 . . . . .	66

ウォッチポイント	68
ウォッチポイントを設定	69
ウォッチポイントを消去	70
ウォッチポイントを表示	70
データの表示、変更	71
局所変数を表示する	72
大域変数を表示する	72
新しいウィンドウへデータを入れる	73
データ型を表示する	74
データを異なるフォーマットで表示する	74
データを異なる型で表示する	75
変数の値を変更する	76
Expression ウィンドウを使う	78
メモリダンプを表示する	78
アドレスを指定してメモリを表示する	79
プロセッサレジスタを表示する	80
ソースコードを編集	81
 第 5 章 評価式	 83
評価式の概要	83
評価式の翻訳	83
Expression ウィンドウの評価式	83
ブレークポイントウィンドウの評価式	84
メモリウィンドウの評価式	85
評価式を使う	85
特別な評価式の機能	85
評価式の制限	86
評価式の例	87
評価式のシンタックス	88
 第 6 章 デバッガの環境設定	 93
環境設定の概要	93
MW Debug の環境設定パネル	93
Settings 環境設定パネル	93
Display 環境設定パネル	95
Symbolics 環境設定パネル	97
Program Control 環境設定パネル	99
Win32 Settings 環境設定パネル	102
Java Settings 環境設定パネル (Windows)	102
Runtime Settings 環境設定パネル (Windows)	103
デバッガのターゲット設定パネル	104

Target Settings 設定パネル	104
x86 Exceptions 設定パネル ( Windows )	105
<hr/>	
第 7 章 デバッガのメニュー	107
デバッガメニューの概要	107
File メニュー	107
Edit メニュー	109
Control メニュー	110
Data メニュー	112
Window メニュー	117
Help メニュー ( Windows )	119
Apple メニュー ( Mac OS )	119
<hr/>	
第 8 章 トラブルシューティング	121
トラブルシューティングの概要	121
一般的な問題	121
デバッガ起動時の問題	122
デバッガが起動しない	122
デバッグできない	122
起動時のエラー ( Mac OS )	123
起動が遅い	123
デバッガ実行時の問題 / クラッシュ	123
プロジェクトをデバッガなしで実行するとクラッシュする	123
ブレークポイントの問題	124
ステートメントにブレークポイントを設定できない	124
ブレークポイントが反応しない	125
変数の問題	125
変数が変化しない	125
変数に誤った値が割り当てられる	126
奇妙な変数	127
奇妙なデータ型	127
データ型が認識されない	128
Expression ウィンドウの『未定義の識別子』	128
ソースファイルの問題	129
ソースコードが表示されない	129
ソースファイルの修正日時	129
プロジェクトでソースコードを共有	130
Pascal プロジェクトの ANSI C コード	130
デバッガのエラーメッセージ	130



## 第 1 章 紹介

CodeWarrior デバッガマニュアルへようこそ。

注意： このマニュアルよりも古いバージョンの IDE が出荷された場合、ここで述べる新機能は利用できません。[『新機能』\(p10\)](#) で新機能をご確認ください。ツールをアップデートするパッチの情報については、Metrowerks の Web サイト <http://www.metrowerks.com> をご覧ください。

### このマニュアルの概要

デバッガは、プログラムの実行をコントロールするアプリケーションです。デバッガを使うと、プログラム実行中にオブジェクトコードの動作を追いながら、プログラムの問題点を見つけることができます。デバッガは、プログラムを 1 ステップずつ実行したり、希望する箇所で実行を停止したりすることができます。デバッガがプログラムを停止した後、変数やメモリの値を見たり、変更したり、関数コールのチェーンを確認したり、プロセッサのレジスタ内容を検証することができます。

このマニュアルは CodeWarrior 統合デバッガ ( Metrowerks CodeWarrior 開発環境が提供するソースレベルデバッガ ) について説明します。このデバッガはサポートするすべての言語 ( C、C++、Pascal、Java、アセンブリ言語 )、およびオペレーティングシステムのいずれにおいても動作します。このマニュアルでは、CodeWarrior 統合デバッガのことを『CodeWarrior デバッガ』または単に『デバッガ』と呼びます。

このマニュアルではすべてのプラットフォームに共通するデバッガの機能について説明します。ターゲットによって異なる機能、またはインプリメントされていない機能があります。ターゲットに特有の機能についてはそれぞれの『Targeting』マニュアルを参照してください。

ここでは次の項目について説明します。

[Metrowerks の西暦 2000 年問題対応](#)

[リリースノートについて](#)

[マニュアルの表記規則](#)

[新機能](#)

[システムの必要条件](#)

[MW Debug をインストール](#)

[出発点](#)

## [さらに学ぶために](#)

## Metrowerks の西暦 2000 年問題対応

ライセンス契約に基づいて Metrowerks が提供する製品は、ホストまたはターゲットオペレーティングシステムによって提供される日付データを利用して内部プロセスの日付データ（ファイル修正日など）を処理しています。ゆえに、西暦 2000 年問題から生じる製品のオペレーションは、ホストまたはターゲットオペレーティングシステムの西暦 2000 年問題対応によるものです。Microsoft 社、Sun Microsystems 社、Apple Computer 社などの西暦 2000 年問題についての文書をお読みください。Metrowerks 製品自体は日付データを処理しないため、西暦 2000 年問題とは無関係です。

詳細は、<http://www.metrowerks.com/about/y2k.html> をご覧ください。

## リリースノートについて

CodeWarrior デバッガを使う前に、最新のデバッガに添付されているリリースノートを必ず参照してください。リリースノートには、重要な情報（新しい機能、バグ修正、最新の情報など）が書かれています。

## マニュアルの表記規則

ここではマニュアルの表記規則を説明します。

### 表記について

特定の情報を表すスタイルについて説明します。

注意、警告、ヒント、および初心者用のヒント

「注意」は、重要な事実を言い換えたり、自明ではない事実に注意を向けます。

「**警告**」は、実行すると取り返しのつかないものを注意したり、発生する可能性のあるエラーを知らせます。

「ヒント」は、CodeWarrior デバッガをより活用するためのヒントです。

「初心者」は、プログラミングの初心者が用語や概念をよりよく理解できるようにします。

### 書体の規則

異なる書体（Courier という書体です）のテキストは、ファイル名やフォルダ名、コード、またはコンピュータのハードディスク上で見られるその他の項目を示します。

CodeWarrior のメニューにある項目は括弧付き（『Open』など）で示します。

下線付きの青色のテキストは（例：『[このマニュアルの概要](#)』）はオンラインドキュメント（Adobe Acrobat など）でのハイパーテキストを示します。これをクリックすると該当ページへジャンプします。



## ホストについて

CodeWarrior は以下のホストプラットフォームおよびオペレーティングシステム上で動作します。このマニュアルでは、オペレーティングシステムに関係なく、プラットフォームの名前をホスト名として使用します。

CodeWarrior は以下のオペレーティングシステム上で動作します。

Windows : Win32 に準拠するデスクトップバージョンの Windows オペレーティングシステム ( Windows 95 や Windows NT など )

Mac OS : デスクトップバージョンの Mac OS ( System 7.1 以降 )

Solaris : バージョン 2.5.1 以降の Solaris

## 画面図について

各ホスト (『[ホストについて](#)』(p9)) のビジュアルインターフェースはほぼ同じです。ダイアログやウィンドウなどのインターフェースの画面図は色々なホストのものを使用しますが、異なるホスト上で CodeWarrior を使っている場合でも理解できるはずです。

ホストに固有のダイアログやウィンドウについては、各ホストにおける画面図を記載します。例えば、Windows と Mac OS 上では大幅にダイアログが異なる場合などは両方の図を載せています。


## キーボードについて

CodeWarrior のキーボードショートカットはあるプラットフォームのものとよく似ています。しかし、プラットフォーム間でキーボードやショートカットが異なります。例えば、Windows マシンでは主に Alt キー、Mac OS マシンでは Option キーを使います。

このため、CodeWarrior のドキュメントでは以下のようにキーを表記します。

Enter/Return : Windows では Enter キー、Mac OS では Return キーです。これは『改行』または『行末』キーです。テンキーパッドの Enter キーとは違いますが、ほとんどの場合同様に働きます。

Backspace/Delete : Windows では Backspace キー、Mac OS では Delete キーです。CodeWarrior ではこれらのキーの使い方はほぼ同じです。このキーは ( テキスト編集においては ) 挿入ポイントの左側にある文字を削除します。

Ctrl/Command : Windows では Ctrl ( control ) キー、Mac OS では Command キー (  ) です。CodeWarrior ではこれらのキーの使い方はほぼ同じです。

Alt/Option : Windows では Alt キー、Mac OS では Option キーです。CodeWarrior ではこれらのキーの使い方はほぼ同じです。

例えば、『Enter/Return キーを押してください』、『Function ポップアップメニューを Alt/Option + クリックして関数をアルファベット順で表示する』などの説明があります。指示通りのキーを使ってください。





複数のキーを使うショートカットは+ マークで示します。例えば Shift + Alt/Option + Enter/Return とある場合、Windows 上では Shift、Alt、Enter キーを同時に押します。Mac OS 上では Shift、Option、Return キーを同時に押します。

クロスプラットフォームでは複雑なキーボードショートカットがあります。この場合、ホストプラットフォームのキーボードショートカットの説明をお読みください。

Solaris ユーザーへの注意

Solaris をホストとする CodeWarrior IDE は Mac OS と同じキー ( Shift、Command、Option、Control ) を使います。Key Bindings 環境設定パネルでも Mac OS シンボルでキーを表しています。[表 1.1](#) にデフォルトのショートカットのキーマッピングとシンボルを示します。Solaris 上では、キーボード上のあらゆるキーに修飾キーを割り当てることができます。Info メニューの『Keyboard Preferences』を選択してデフォルトの修飾キーを変更します。

表 1.1 Mac OS と Solaris の修飾キー

シンボル	Mac OS	Solaris
	Command キー	Meta キー
	Option キー	Alt キー
	Shift キー	Shift キー
	Control キー	Control キー

新機能

CodeWarrior デバッガの新機能について説明します。

CodeWarrior IDE と統合デバッガ

CodeWarrior デバッガを CodeWarrior IDE に統合したことにより、ソースコードのプログラミングとデバッグに以下の利点をもたらします。

メモリ使用量の減少：一つのアプリケーションとして実行されるため、必要なメモリが少なくなります。

生産性の向上：コードのステップ実行やブレークポイントを設定するために IDE とデバッガを切り替える必要がないため、時間の節約につながり、生産性を向上します。

統合デバッガは、x86、PowerPC、68K、Java を完全にサポートします。各プラットフォーム用のデバッガは不要になります。

統合デバッガを有効にすると、Project メニューから『Enable Debugger』を選択するだけで統合デバッガを使用できます。いつでもプログラムを一時停止してブレークポイントを設定したり、変数やメモリを見たり、関数にステップイン、アウトすることができます。

注意： 統合デバッガを含まないバージョンの CodeWarrior IDE では、独立したアプリケーションである MW Debug か、サードパーティのデバッガでデバッグを行います。あるターゲットをデバッグするには、各プラットフォームの『Targeting』マニュアルを参照してください。

独立したデバッガを使うには、デバッガを起動してから『Debug』を選択してください。

## システムの必要条件

ほとんどのバージョンの CodeWarrior IDE にはデバッガが統合されています。CodeWarrior IDE の Project メニューに『Enable Debugger』か『Disable Debugger』コマンドがあれば、統合デバッガを使用できます。それ以外は、独立したデバッガをインストールする必要があります。

MW Debug (独立したデバッガ) は統合デバッガを持たない CodeWarrior IDE に含まれています。以下は MW Debug の必要条件です。

### Windows

CodeWarrior デバッガは、486、Pentium (TM) 以上のプロセッサ、16MB の RAM および 5MB のディスクスペースを必要とします。CodeWarrior デバッガは、Windows 95 または Windows NT 4.0 オペレーティングシステムで動作します。

最高のパフォーマンスを得るためには、Pentium (TM) 以上のプロセッサに最低で 24MB の RAM を装着し、Windows NT 4.0 上で使用することをお勧めします。

### Mac OS

CodeWarrior デバッガは Motorola 68020 プロセッサ以上、または PowerPC 601 以上のプロセッサを必要とします。CodeWarrior デバッガは約 2MB のハードディスク容量と 1.5MB の RAM を必要とします。

CodeWarrior デバッガは System 7.1 以降 (68K Macintosh) または System 7.1.2 以降 (Power Macintosh) の OS、Color QuickDraw を必要とします。68K システムでは CFM-68K 機能拡張が必要です。

Watchpoint 機能は、Motorola 68020 以上のプロセッサでは仮想メモリをオンの設定で、または PowerPC 601 以上のプロセッサ (仮想メモリはオン、オフいずれでも) で動作します。Watchpoint 機能は System 7.5 以降の OS を必要とします。[『ウォッチポイント』\(p68\)](#) を参照してください。

### Solaris

MW Debug には Sun SparcStation または Sparc ベースのマシンが必要です。最小 32MB の RAM、CD-ROM ドライブ、40MB のディスクスペース、Network Information Service、X11 サーバ (Open Windows v3.3 推奨)、vX11r5 以降の Window Manager、1.2.2 以降の Motif が必要です。

## MW Debug をインストール

CodeWarrior IDE にデバッガが統合されている場合、MW Debug のインストールは不要です。MW Debug デバッガは統合デバッガと同じ機能を提供する独立したアプリケーションです。詳細は『[システムの必要条件](#)』(p11) を参照してください。

CodeWarrior コンパイラやターゲットプラットフォームに関係なく、MW Debug は一つです。MW Debug は CodeWarrior IDE と異なるアプリケーションですが、CodeWarrior IDE と連動して動作します。

CodeWarrior のインストーラを使って CodeWarrior IDE をインストールすると、デバッガと必要なツールも同時にインストールされます。CodeWarrior デバッガは、Debugger Plugins フォルダと同じディレクトリに入れる必要があります。そうでない場合、デバッガは動作しません。

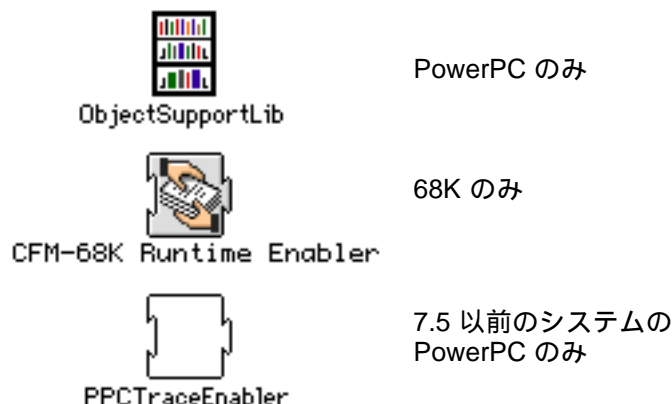
必要なファイルをすべてインストールし、インストールで問題を起こさないようにするために、デバッガのインストールには CodeWarrior インストーラを使うことを強くお勧めします。

### Mac OS

MWDebug と Debugger Plugin のフォルダは (Helper Apps) フォルダに入れる必要があります。これ以外の場所では、プロジェクトから直接起動することができません。

MW Debug を使うために、いくつかの機能拡張ファイルをシステムフォルダの機能拡張フォルダにインストールする必要があります。インストーラは、自動的に機能拡張フォルダに必要な機能拡張ファイルをインストールします。これらの機能拡張を手作業でインストールまたは取り外した場合は、変更を有効にするためにコンピュータを再起動する必要があります。デバッガを使用する前に、[図 1.1](#) に示すように、カレントターゲットとプラットフォームに合った正しい機能拡張ファイルがインストールされていることを確認してください。

図 1.1 MW Debug に必要なファイル (Mac OS)



ObjectSupportLib : PowerPC オブジェクトコードをデバッグするときに必要な共有ライブラリです。

CFM-68K Runtime Enabler：デバッガの共有ライブラリを使うときに必要です。共有ライブラリを使わないデバッガも Tools CD に含まれています。

PPCTraceEnabler：System7.5 以前の MacOS において PowerPC コードをデバッグするときに必要です。

## 出発点

[この章の概要](#)：デバッガのインストールと実行の方法、シンボルファイルとは何かを説明します。

[ウィンドウの概要](#)：デバッガの各種ウィンドウについて説明します。

[デバッガの概要](#)：デバッガの基本的な機能と使い方を説明します。

[評価式の概要](#)：デバッガ内での評価式の使い方を紹介します。

[環境設定の概要](#)：デバッガの環境設定の方法について説明します。

[デバッガメニューの概要](#)：デバッガのメニューを紹介します。

[トラブルシューティングの概要](#)：デバッガに関してよく質問される内容（FAQ：frequently asked questions）とその解決方法を紹介します。

CodeWarrior のデバッガを初めて使う方、インストール手順について疑問がある方、またはシンボルファイルについて知らない方は、『[この章の概要](#)』（p15）をご覧ください。デバッガのインタフェースについては、『[ウィンドウの概要](#)』（p21）をご覧ください。

プログラムの実行方法、ブレークポイントの設定方法、変数の変更方法の詳細は『[デバッガの概要](#)』（p47）と『[評価式の概要](#)』（p83）をご覧ください。

デバッガのメニューリファレンスは『[デバッガメニューの概要](#)』（p107）をご覧ください。

デバッグの熟練度にかかわらず、デバッガを使っていて何か問題に遭遇したときは、『[トラブルシューティングの概要](#)』（p121）をご覧ください。よく遭遇する問題点とその解決方法について説明しています。

## さらに学ぶために

既に基本的なデバッグ操作について知っているが、特別なコードのデバッグについて知りたい場合、該当する『Targeting』マニュアルを参照してください。





## 第 2 章 はじめに

この章では、デバッグの準備、シンボルファイルについて説明します。この後の章でデバッグの機能を解説します。

### この章の概要

この章には、デバッグを効率的に使うために必要な背景となる情報が含まれています。ここでは、次の内容を解説します。

[デバッグの準備](#)

[デバッグを起動](#)

[シンボルファイル](#)

### デバッグの準備

あるターゲット用に生成された CodeWarrior プロジェクトのファイルをデバッグするには、ビルドターゲットとそれに含まれる各ファイルの両方に、デバッグに必要な設定がなされていなければなりません。これによってデバッグに必要なシンボル情報が生成されます。

ここでは以下の内容について説明します。

[デバッグのためにターゲットを準備](#)

[デバッグのためにファイルを設定](#)

[シンボル情報を生成](#)

#### デバッグのためにターゲットを準備

デバッグのためにビルドターゲットを準備するには、CodeWarrior IDE の Project メニューの『Enable Debugger』を選択します。デバッグが可能になると Project メニューの『Enable Debugger』は『Disable Debugger』へ切り替わります。デバッグ機能をオフにするには、『Disable Debugger』を選択してください。メニューは『Enable Debugger』に戻ります。

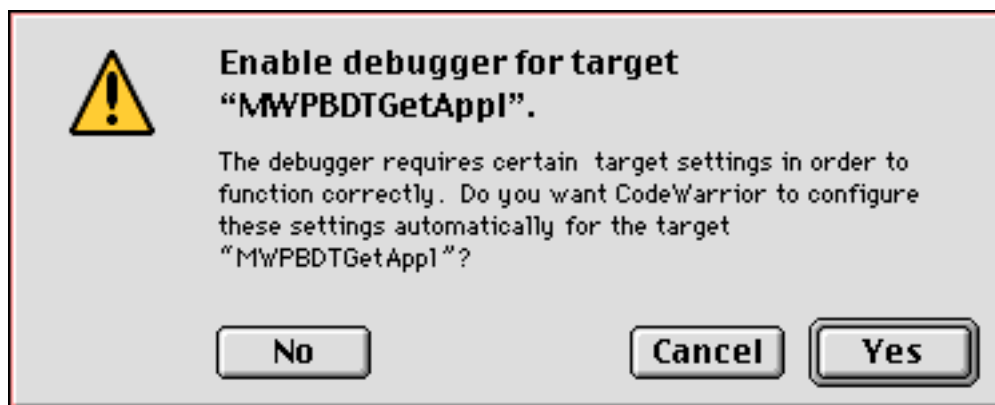
『Enable Debugger』は、プロジェクトウィンドウと設定パネルにあるデバッグに関する各項目を設定し、デバッグ情報を生成するようにコンパイラとリンカを設定します。コンパイラおよびリンカはプログラムに特別なコードを追加し、シンボルファイルを生成します。シンボルファイルには、ソースレベルのデバッグに必要な情報が含まれます。

リンカとプロジェクトの設定の詳細は『CodeWarrior IDE User Guide』をご覧ください。

注意： シンボルファイルにより、デバッガはソースコード内で使われる関数と変数名（シンボル）を追跡できます。詳しくは、[『シンボルファイル』\(p19\)](#)をご覧ください。

『Enable Debugger』を選択すると警告が表示されることがあります（[図 2.1](#)）。[ Yes ] をクリックし、デバッグ用の設定を適用してください。

図 2.1 Enable Debugger の設定を適用する



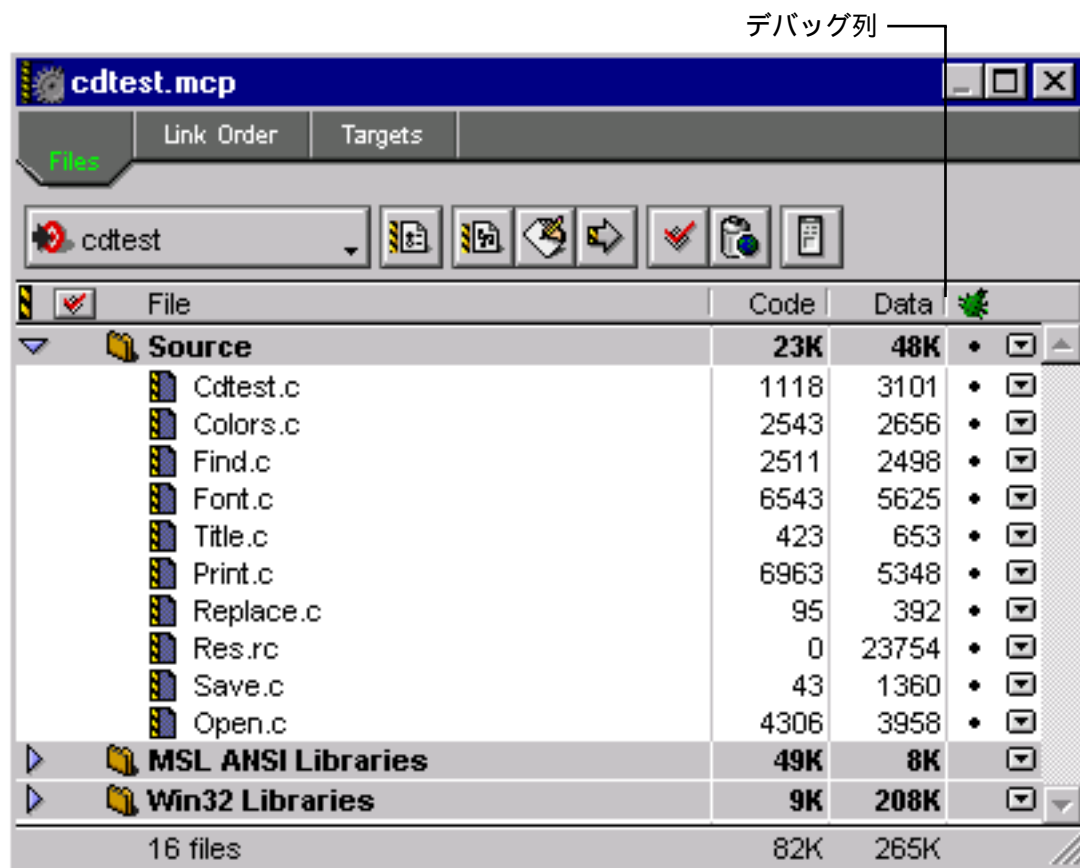
### デバッグのためにファイルを設定

カレントビルドターゲットに対してデバッグの設定を行った後、個々のファイルのデバッグフラグが設定されていることを確認してください。プログラムをデバッグする際は、通常すべてのソースファイルに対してデバッグフラグをオンにします。

CodeWarrior IDE のプロジェクトウィンドウには、デバッグ列があります（[図 2.2](#)）。ファイルの右にあるデバッグ列の点は、そのファイルに対してデバッグが可能であることを示します。黒い点がないとき、そのファイルはデバッグできません。グループ名のデバッグ列に黒い点があるとき、そのグループに属するすべてのファイルがデバッグ可能であることを示します。点がないとき、そのグループに属する一部のファイルがデバッグできません。



図 2.2 プロジェクトウィンドウのデバッグ設定



ファイルのデバッグ機能をオンまたはオフにするには、ファイル名の横のデバッグ列をクリックします。グループ名の横のデバッグ列をクリックすると、グループに属するすべてのファイルのデバッグ機能が、オン / オフに切り替わります。ファイルがデバッグ不可能な場合（ファイルがソースファイルではないなど）、そのファイルについてはデバッグ機能をオンにできません。

### シンボル情報を生成

シンボル情報を生成するためには、カレントビルドターゲットとソースファイルの両方を正しくデバッグ用に設定しなくてはなりません。詳細は『[デバッグのためにターゲットを準備](#)』(p15)、『[デバッグのためにファイルを設定](#)』(p16)を参照してください。

カレントターゲットとソースファイルの設定を終えたら、Project メニューの『Make』を選択します。最終的なコードがコンパイルおよびリンクされます。

コンパイルおよびリンクについての詳細は『CodeWarrior IDE User Guide』、およびそれぞれの『Targeting』マニュアルを参照してください。

## デバッガを起動

デバッガは通常 IDE から直接起動します。または独立したアプリケーションとして MW Debug だけを起動することもできます。ターゲットであるチップやオペレーティングシステム、生成するコードの種類によっていずれかの方法で起動することができます。

MW Debug を起動すると、シンボルファイルの位置を尋ねられます。

ターゲットやプロジェクトによっては IDE からデバッガを起動できます。

ここではデバッガを起動する方法について説明します。

[MW Debug を IDE から起動 \( Mac OS \)](#)

[MW Debug を直接起動](#)

いずれもシンボルファイルを必要とします。シンボルファイルの生成については『[デバッグの準備](#)』(p15) を参照してください。

### 統合デバッガを使う

通常は CodeWarrior IDE に統合されている CodeWarrior デバッガを使います。

CodeWarrior デバッガを起動するには、デバッガを使用可能にします。デバッガが使用不可の場合、Project メニューの『Debug』コマンドを選択する前に『Enable Debugger』を選択してください。

デバッガが使用可能になると、IDE の Project メニューの『Run』コマンドが『Debug』に変わります。『Debug』コマンドを選択すると IDE が直接デバッガを起動します。デバッガは自動的にシンボルファイルを開きます。またはシンボルファイルの位置を尋ねます。

IDE では、実行可能コード ( アプリケーションなど ) を生成するターゲットに対してのみ『Debug』コマンドが使用できます。

### MW Debug を IDE から起動 ( Mac OS )

デバッガが統合されていないバージョンの CodeWarrior IDE では、MW Debug というデバッグアプリケーションを使います。

これは IDE とは別のアプリケーションで、CodeWarrior IDE と同じディレクトリに MW Debug を置いてください。MW Debug は必ずバックグラウンドで動作していて、ソースファイルを開いておく必要があります。

デバッガが使用不可の場合、『Debug』コマンドを選択する前に『Enable Debugger』を選択してください。

デバッガが使用可能になると、IDE の Project メニューの『Run』コマンドが『Debug』に変わります。このコマンドは、プロジェクトをコンパイルおよびリンクし、その後デバッガを通して起動します ( 詳細は『[デバッグの準備](#)』(p15) を参照してください )。

外部デバッガでアプリケーションをデバッグするには、『Switch to MW Debugger』を選択してください。

IDE では、実行可能コード（アプリケーションなど）を生成するターゲットに対してのみ『Debug』コマンドが使用できます。

ライブラリや共有ライブラリを生成するターゲットに対してもデバッグは可能ですが、この場合そのコードを使うアプリケーションを起動してデバッガを直接起動する必要があります。『[MW Debug を直接起動](#)』（p19）を参照してください。

## MW Debug を直接起動

デバッガが統合されていないバージョンの CodeWarrior IDE では、MW Debug というデバッグアプリケーションを使います。

MW Debug は独立したアプリケーションなので、普通のアプリケーションのようにデバッガを直接起動することもできます。デバッガで作業を行う場合は、必ずシンボルファイルが必要とします。デバッガを起動する方法は、次の三つがあります。

シンボルファイルをダブルクリックしてください。

デバッガアイコンをダブルクリックしてください。シンボルファイル名を入力するための標準ダイアログが表示されます。

シンボルファイルをデバッガアイコンにドラッグ&ドロップして起動してください。

デバッガを直接起動することは頻繁にあります。IDE では起動できないターゲットやコードがあります。例えば、アプリケーションのプラグインはそれのみでは実行できません。また CodeWarrior IDE にはそのプラグインを使うアプリケーションも不明です。

以下に、プラグインをデバッグする際の典型的な手順を示します。

1. デバッガを直接起動する。
2. プラグインのシンボルファイルを開く。
3. コードにブレークポイントを設定する。  
(詳細は『[ブレークポイント](#)』（p63）を参照してください。)
4. プラグインを使うアプリケーションを起動する。
5. アプリケーションにプラグインのコードを呼び出させる。

プラグインのコード実行がブレークポイントに達したとき、デバッガにコントロールが移るのでプラグインのコードをデバッグできます。

## シンボルファイル

プロジェクトのシンボルファイルには、プロジェクトをデバッグするためにデバッガが必要とする情報が含まれています。ルーチン名や変数名（シンボル）、ソースコードのどの位置にそれらのシンボルがあるか、オブジェクトコードのどこにそれらがあるか、という情報も含まれています。

デバッガはこれらの情報を使ってオブジェクトコードに対応するソースコードを表示します。デバッガを停止したときにソースが表示されます。

対応するアセンブリ言語のインストラクションとメモリアドレスを見ることもできます。[『ソースコードをアセンブラで見る』\(p27\)](#) を参照してください。

CodeWarrior はその他のターゲットのシンボルフォーマットもサポートします。

フォーマット	ターゲット
CodeView	Win32
DWARF	組み込みシステム
SYM	Mac OS

プロジェクトとソースファイルの設定、シンボルファイルの生成方法については [『デバッグの準備』\(p15\)](#) を参照してください。

コンパイラやリンカの設定を含むシンボル情報の生成についての詳細は『CodeWarrior IDE User Guide』を参照してください。

ターゲットに特有のシンボル情報についてはそれぞれの『Targeting』マニュアルを参照してください。



## 第 3 章 デバッガのウィンドウ

この章では CodeWarrior デバッガの各種ユーザーインターフェースの構成要素について説明します。

### ウィンドウの概要

この章では CodeWarrior デバッガの各種ウィンドウ、表示欄、および表示される項目について説明します。このマニュアルの後の章には、デバッガの各部分の動作と目的の理解が必要です。この章では、次の項目を説明します。

[プログラムウィンドウ](#)

[ブラウザウィンドウ](#)

[Expression ウィンドウ](#)

[Breakpoint ウィンドウ](#)

[Watchpoint ウィンドウ](#)

[Log ウィンドウ](#)

[変数ウィンドウ](#)

[配列ウィンドウ](#)

[メモリウィンドウ](#)

[レジスタウィンドウ](#)

[プロセスウィンドウ](#)

### プログラムウィンドウ

デバッガがシンボルファイルを開くとき、プログラムウィンドウ ([図 3.1](#)) を開きます。

プログラムウィンドウは、現在実行中の関数を含むソースファイルに関するデバッグ情報を表示します。このウィンドウには図に示す四つの欄があります。

[スタッククロール欄](#)

[変数欄](#)

[デバッガのツールバー](#)

[ソース欄](#)

欄の境界線の間をクリックまたはドラッグして、各欄の大きさを自由に変更することができます。アクティブ状態の欄は太い枠に囲まれています。欄を切り替えるには、Tab キーを使います。

タイプアヘッド選択は、スタッククロール欄および局所変数欄で有効です。アクティブ欄の中で移動するには、矢印キーか Tab キーを使います。

ソース欄の一番下の水平スクロールバーの左側に、その他のコントロール項目があります。

関数ポップアップメニュー

現在の行番号

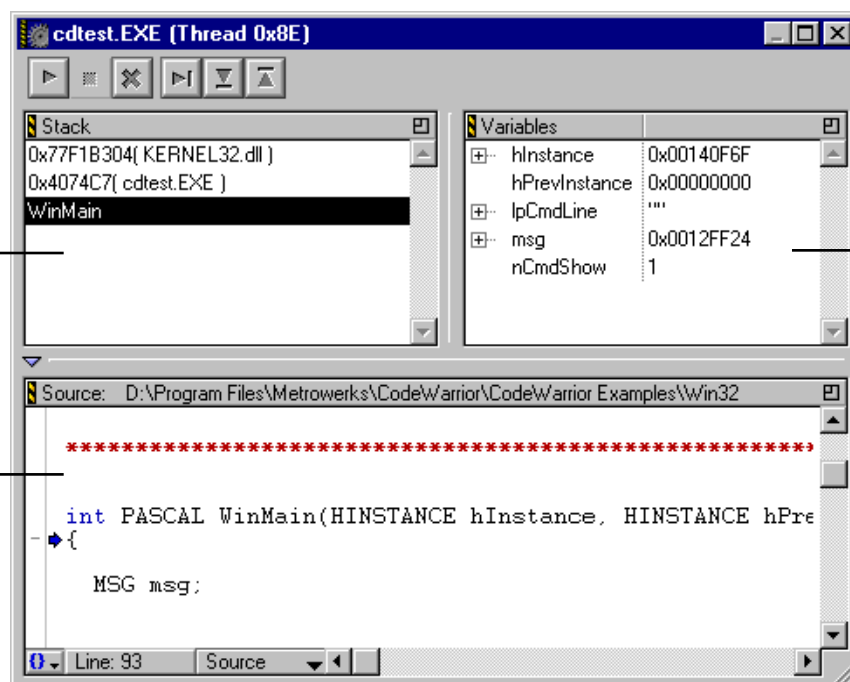
ソースポップアップメニュー

ブラウザウィンドウの内容の詳細は、[『ブラウザウィンドウ』\(p29\)](#) もご覧ください。

図 3.1 プログラムウィンドウの各部

ソース欄に関数を表示するには、スタッククロール欄の関数の名前をクリックする。

ソース欄には現在実行中のソースコードとブレークポイントが表示される。



局所変数や参照している大域変数を表示するには変数欄を使う。

## スタッククロール欄

プログラムウィンドウのスタッククロール欄は、現在コールチェーンで呼び出されているサブルーチンを表示します( [図 3.2](#) )。それぞれの関数は、呼び出し元の関数の下に表示されます。

ハイライトされている関数がウィンドウの一番下のソース欄に表示されます。関数のソースコードを表示するために、スタックロール欄で関数を選択してください。

図 3.2 スタックロール欄

NewBall() の内容を現在、ソース欄に表示している。これは main() と呼ばれる。

main() の内容をソース欄に表示するには、スタックロール欄でその名前をクリックする。



## 変数欄

変数欄 ( [図 3.3](#) ) は、実行中の関数の局所変数を表示します。

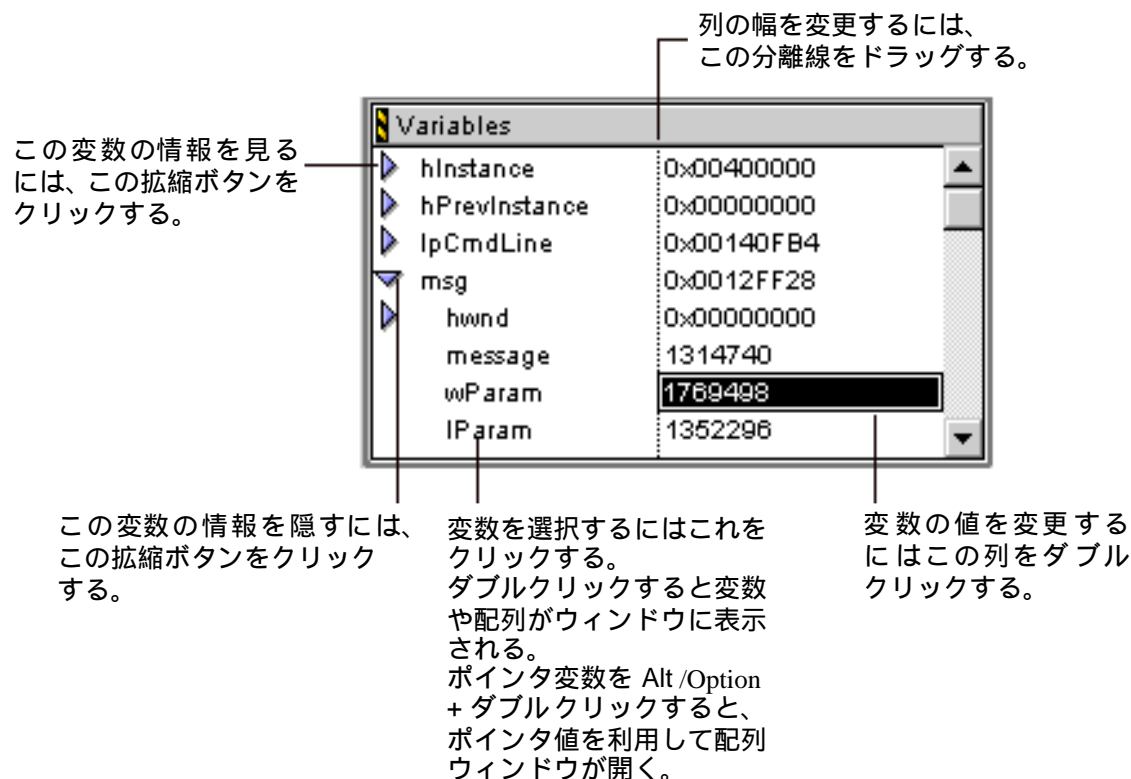
Mac OS :変数欄には、実行中の関数の局所変数、およびその関数が参照している大域変数を表示します。局所変数と大域変数の表示領域は破線で区切られています。

変数欄では、変数をアウトライン形式で表示します。アイテムの横にあるツリーコントロール ( Windows ) か、拡張ボタン ( Mac OS ) をクリックすると、変数の中のエントリが表示されたり、隠されたりします。

例えば [図 3.3](#) では、変数 msg の下向きのボタンをクリックすると、メンバ変数が隠れます。再びそのボタンをクリックすると、C の構造体のメンバ変数が表示されます。Ctrl/Option キーを押しながらクリックすると、複数の階層のポインタの変数を見ることができます。ハンドルを拡張して、構造体の型にあてはめてメンバ変数を見ることができます。

[『Expand』 \( p113 \)](#) および [『Collapse All』 \( p113 \)](#) もご覧ください。

図 3.3 変数欄



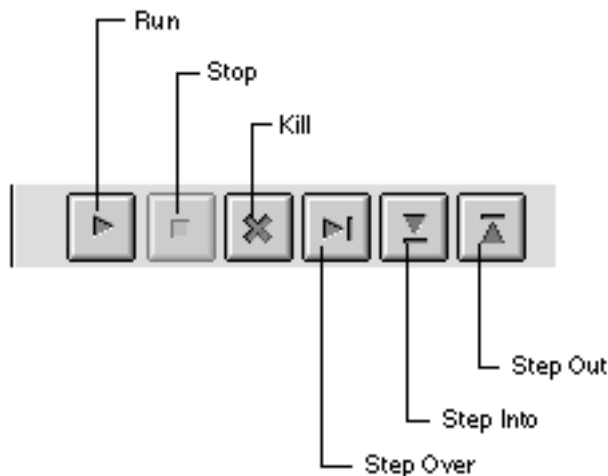
注意： アセンブラコードを表示している場合、変数欄にレジスタやメモリは表示されません。代わりにレジスタウィンドウやFPU レジスタウィンドウ（『[レジスタウィンドウ](#)』(p41)）を使って、CPU や演算コプロセッサのレジスタの値を見てください（FPU を持たないターゲットでは、FPU レジスタウィンドウは利用できません）。

## デバッガのツールバー

ツールバー（[図 3.4](#)）には、Control メニューの『Run』、『Stop』、『Kill』、『Step Over』、『Step Into』、『Step Out』を実行するコマンドのボタンがあります。

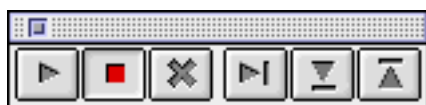


図 3.4 デバッガのプロセスウィンドウのツールバー



Mac OS :MW Debug には小さいフロートツールバーもあります(図 3.5)。Window メニューの『[Show/Hide Toolbar \( Mac OS \)](#)』を選択すると、フロートツールバーが表示 / 非表示されます。

図 3.5 デバッガのフロートツールバー



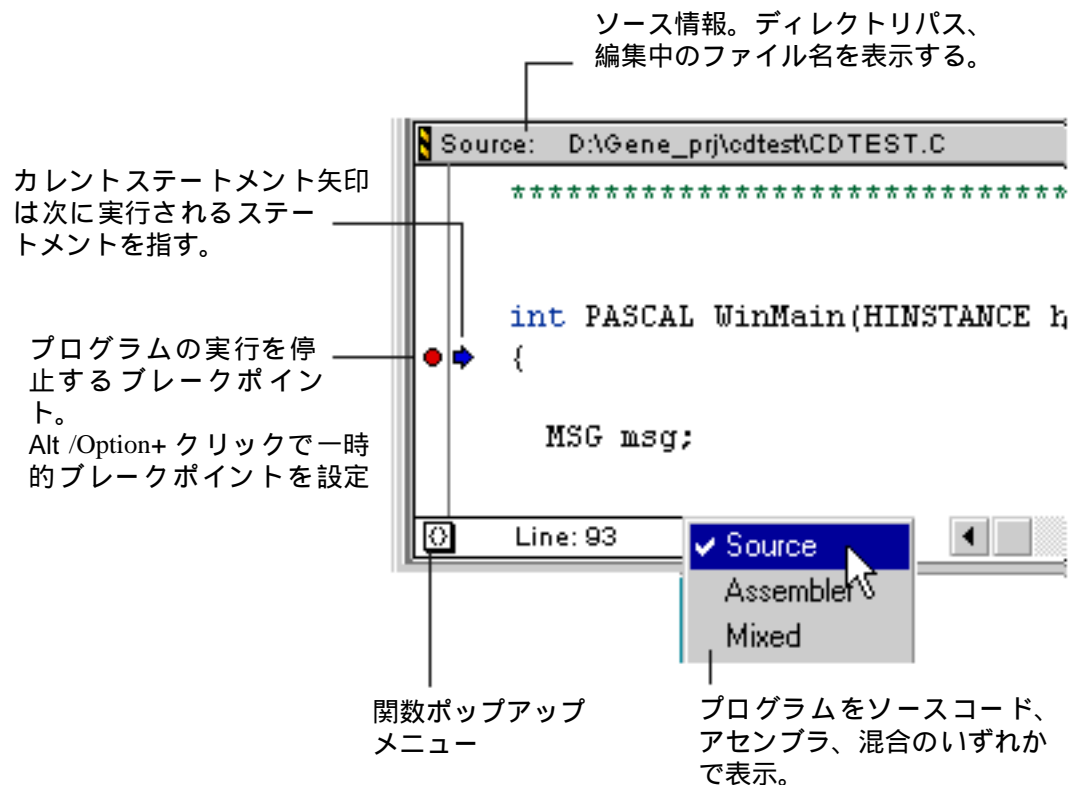
『[デバッガの概要](#)』(p47) を参照してください。

## ソース欄

ソース欄は、現在実行中のソースファイルを表示します。デバッガは、カレントターゲットのソースファイルからコメントやスペースを含むソースコードを持ってきます。ソース欄には、C/C++、Pascal、Java、およびインラインアセンブラコードを CodeWarrior IDE のテキストのフォントや色の設定を使って表示します(図 3.6)。

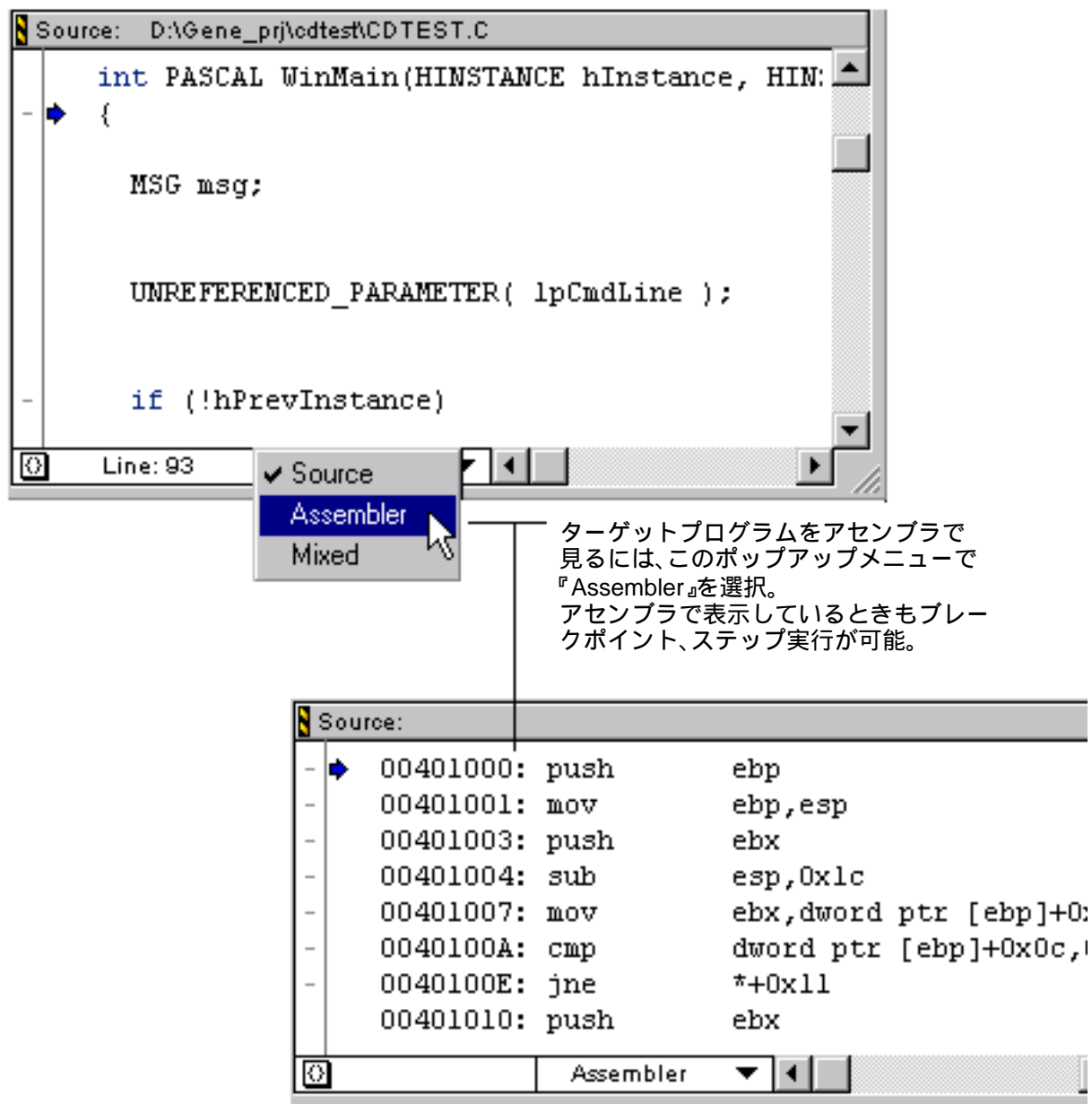
ソース欄では、プログラムのソースコードを一行ずつ実行でき、その過程がステートメント矢印で示されます。カレントステートメント矢印は、次に実行されるステートメントを示しています。

図 3.6 ソース欄 (プログラムウィンドウ)



一行の中に、複数の関数が呼び出されているとき、各関数は別々に1ステップで実行されます。このような場合、すべての関数が実行されてからステートメント矢印が次の行に移動します。矢印は、プログラムカウンタがソースコードの途中にあるときには薄く表示されます。

図 3.7 ソースとアセンブラの表示



### ソースコードをアセンブラで見る

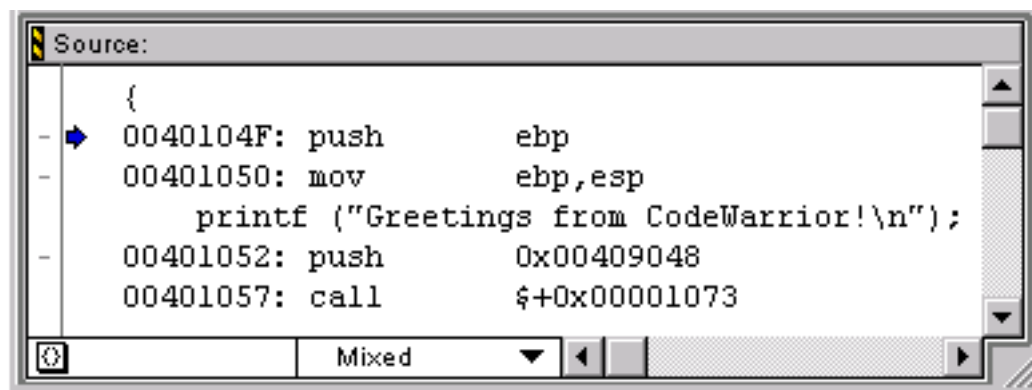
ソースコードをアセンブラで見るときは、プログラムウィンドウの下にあるソースポップアップメニューをクリックしてください。『Assembler』を選択すると、ソース欄にアセンブラコードが表示されます(図 3.7)。アセンブラコードを表示しているときも、ステップ実行や、ブレークポイントの設定ができます。

注意： アセンブラコードを表示しているときは、変数欄にレジスタやメモリは表示されません。レジスタウィンドウやFPU レジスタウィンドウ（『[レジスタウィンドウ](#)』（p41））を使って、CPU や演算コプロセッサのレジスタの値を見てください。（FPU を持たないターゲットではFPU レジスタウィンドウは利用できません）。

#### アセンブラとソースを混合表示

ソースコードとアセンブリ言語を同時に表示するには、プログラムウィンドウ下部のソースポップアップメニューをクリックしてください。『Mixed』を選択すると現在のルーチンのソースコードとアセンブラコードが表示されます（[図 3.8](#)）。アセンブラ命令を生成したソースは、そのアセンブラの前に表示されます。コードを混合で表示するとき、デバッガはアセンブラコードを『ライブ』にします。つまりアセンブラコードにブレークポイントを設定したり、ステップ実行することができますが、ソースコードの行にはブレークポイントは設定できません（[図 3.8](#)）。

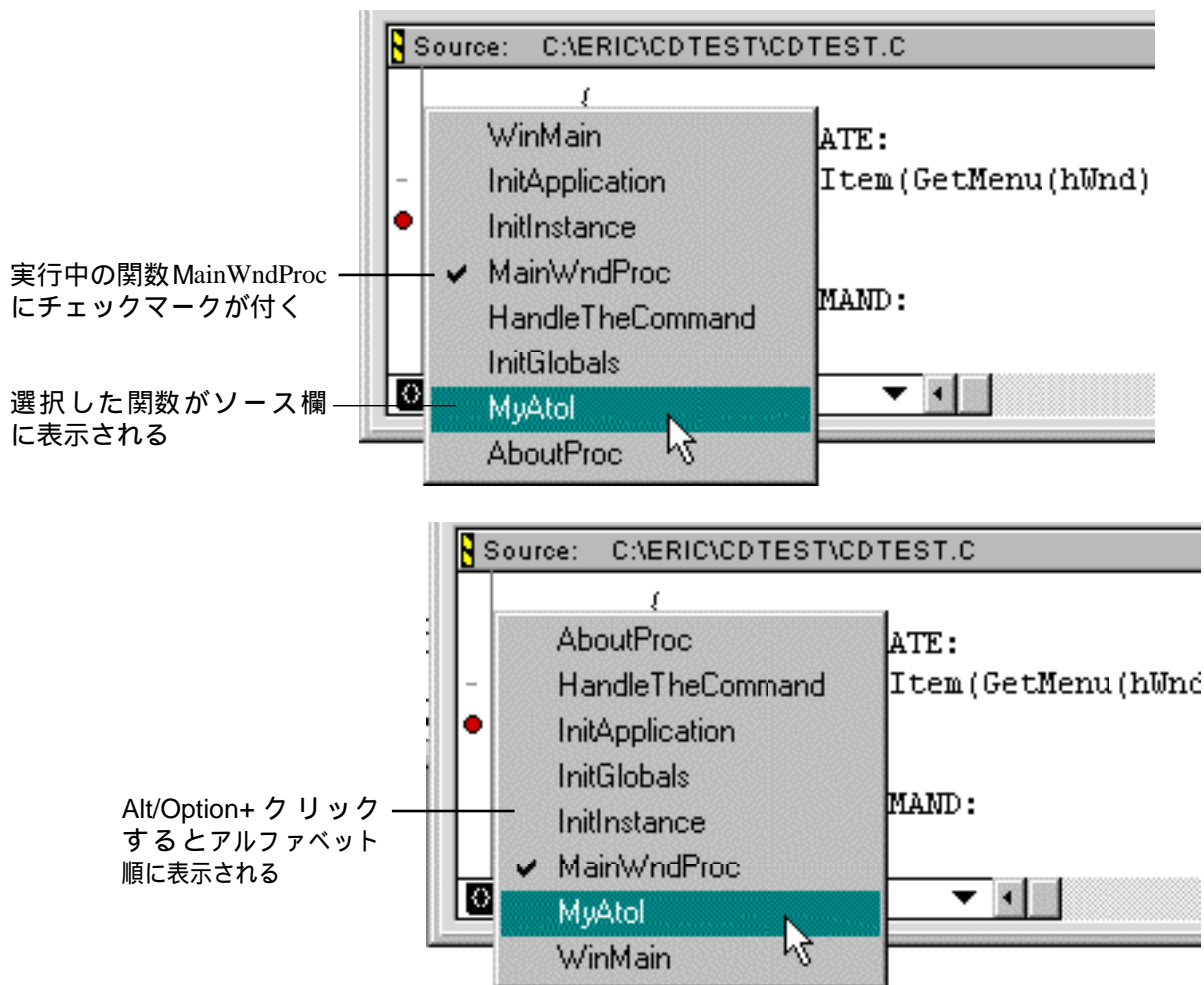
図 3.8 コードの混合表示



コードのソースがない場合、アセンブラを表示します。混合コードを表示しているとき、シンタックスハイライトは使えません。テキストはすべて標準で表示されます。

アセンブラとソースの混合表示は Java 用の MW Debug では使えません。この機能は統合 CodeWarrior デバッガでのみ利用可能です。

図 3.9 関数ポップアップメニュー



#### 関数ポップアップメニュー

ソース欄の左下にある関数ポップアップメニュー（[図 3.9](#)）には、ソース欄で選択したソースファイルで定義されている関数のリストが表示されます。関数ポップアップメニューで関数を選択すると、ソース欄にその関数が表示されます。

関数ポップアップメニューを Alt/Option + クリックすると関数名がアルファベット順に表示されます。

## ブラウザウィンドウ

MW Debug がシンボルファイルを開くとき、プログラムウィンドウとブラウザウィンドウの二つを開きます。二つは似ていますが、細かいところが違います。

ブラウザウィンドウは、プログラムウィンドウに外観も機能もよく似ています（[図 3.10](#)）しかし、表示される情報が異なります。ブラウザウィンドウでは、カレントビルドターゲット

トにあるどのファイルでも選択して見ることができます。一方プログラムウィンドウでは、スタックロール欄で選択された、現在実行中の関数を含むファイルだけを表示できます。また、ブラウザウィンドウを使って、プログラムのすべての大域変数の値を編集できます。プログラムウィンドウでは、コールチェーンで現在アクティブな関数が参照している大域変数しか変更できません。

---

初心者： ブラウザウィンドウと、CodeWarrior IDE で利用できるクラスブラウザを混同しないでください。この二つの外観はよく似ていますが、デバッガのブラウザウィンドウはソースコードのブラウザでありクラスのブラウザではありません。

---

ブラウザウィンドウにも、四つの表示欄があります。

[ファイル欄](#)（左上）

[関数欄](#)（中上）

[大域変数欄](#)（右上）

[ブラウザのソース欄](#)（下）

プログラムウィンドウと同様に、ブラウザウィンドウにもウィンドウの下に、関数ポップアップメニュー、行番号表示、ソースポップアップメニューがあります。またプログラムウィンドウと同様に、境界線をクリックやドラッグすることで、ブラウザウィンドウの各欄のサイズを変更することもできます。Tab キーを入力して、各欄を移動できます。

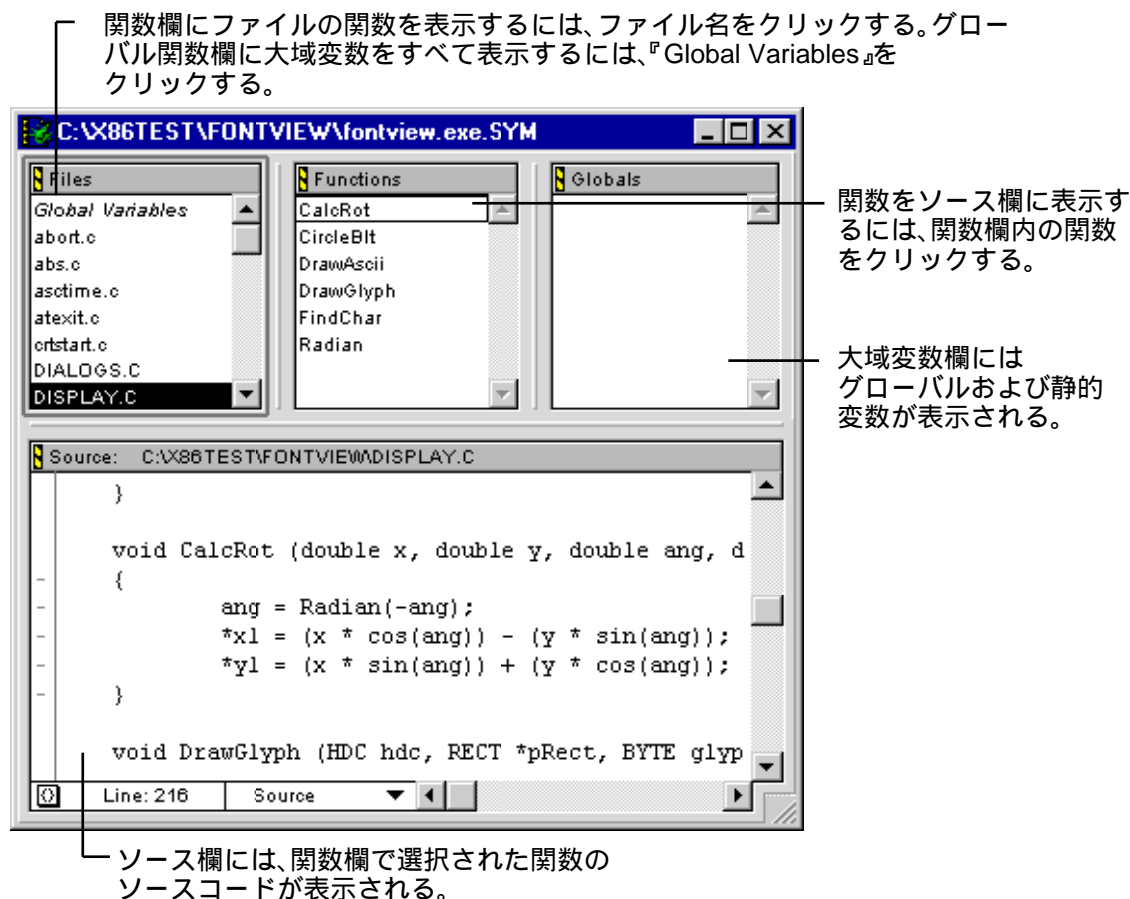
Mac OS :アクティブな欄は太い線で囲まれています。

ファイル欄、関数欄、大域変数欄でタイプahead選択ができます。矢印キーか Tab キーを使って、アクティブな欄内でアイテムをナビゲートすることもできます。

同時に複数のシンボルファイルを開いてデバッグできます。これにより複数のプログラムを一度にデバッグできます。アプリケーションとアプリケーションのプラグインを同時にデバッグするときなどに便利な機能です。

プログラムウィンドウの内容について詳しくは、[『プログラムウィンドウ』\(p21\)](#) もご覧ください。

図 3.10 ブラウザウィンドウ



## ファイル欄

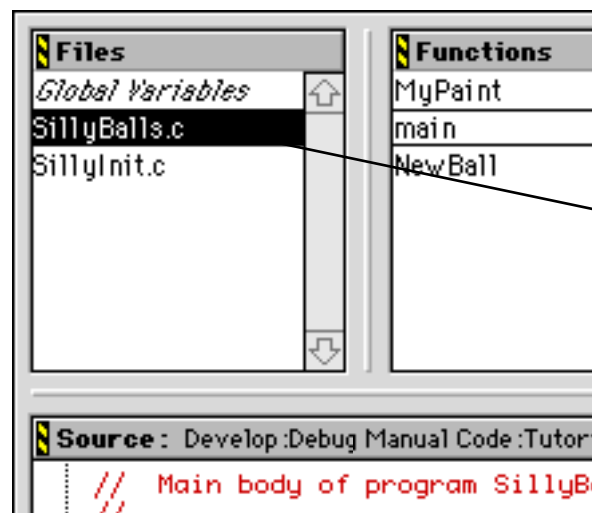
ブラウザウィンドウのファイル欄には、現在デバッグしているカレントターゲットに関連するソースファイルがすべて表示されます(図 3.11)。ファイル欄でファイルを選択すると、そのファイルの関数のリストが関数欄に表示されます。

ファイル欄は、関数欄、ソース欄とともに、プログラムにブレークポイントを設定するために使います。ファイル欄の『Global Variables』を選択すると、プログラムで使われているすべての大域変数が大域変数欄にリストされます。

[『大域変数欄』\(p33\)](#) と [『ブレークポイント』\(p63\)](#) もご覧ください。

図 3.11 ファイル欄

ファイル欄には、ターゲットプログラムのソースファイル、ライブラリファイルおよび大域変数や静的変数が表示される。  
『Global Variables』を選択すると、グローバル欄に大域変数や静的変数が表示される。



SillyBalls.c という  
ハイライトされて  
いるファイルが  
ソース欄に表示さ  
れる。

## 関数欄

ブラウザウィンドウのファイル欄でソースファイルを選択すると、関数欄にはそのファイルで定義されているすべての関数が表示されます。関数名をクリックすると、ウィンドウの下ソース欄がスクロールされてその関数が表示されます。

図 3.12 関数欄

関数欄にはファイル欄で選択されたファイルに含まれているルーチンが表示される。



関数欄で選択された  
ルーチンの内容がソー  
ス欄に表示される。

注意： C++ または Object Pascal でコードを書いている場合、デバッガの Debugger User Guide 環境設定で『Sort functions by method name in browser』を設定すると（『[Settings 環境設定パネル](#)』（p93））関数欄の関数名をアルファベット順に並べて表示します。className::methodName のようなメンバ関数（メソッド）についてもクラス名ではなく、メンバ関数名でソートします。C++ ソースファイルでは、同じクラスのメンバ関数は一つのファイル内にあるため、このソート方法を設定しておく便利です。



## 大域変数欄

ファイル欄で『Global Variables』を選択すると、大域変数欄には、プログラムで使用しているすべての大域変数が表示されます（[図 3.13](#)）。大域変数を見るには、ファイル欄で、そのグローバルが定義されているファイルを選択します。ファイル欄でファイルを選択することにより、静的変数も表示できます。静的変数も大域変数欄に表示されます。

図 3.13 大域変数欄



### 大域変数のみ別ウィンドウに表示

大域変数を別のウィンドウに表示するには、大域変数欄で大域変数名をダブルクリックします。その変数の名前と値を含む新しい変数ウィンドウが表示されます。表示したい変数を大域変数欄で選択して、Data メニューの『View Variable』または『View Array』を選択することもできます。別ウィンドウに表示された大域変数も、大域変数欄と同じように表示または編集できます。大域変数を Expression ウィンドウに追加することもできます。

[『変数ウィンドウ』\(p37\)](#)、[『配列ウィンドウ』\(p38\)](#) および [『Expression ウィンドウ』\(p35\)](#) もご覧ください。

デバッグを行っている間、大域変数のウィンドウは開いています。大域変数、またはグローバル配列ウィンドウを閉じるときは、クローズボックスをクリックしてください。

[『Close All Variable Windows』\(p118\)](#) もご覧ください。

## ブラウザのソース欄

ブラウザウィンドウのソース欄には、ファイル欄（[図 3.14](#)）で選択したソースファイルが表示されます。ファイル欄に表示されるファイルにブレークポイントを設定するには、ソース欄を使います。しかし、ブラウザウィンドウのソース欄には現在実行されているステートメントが表示されていないことに注意してください。現在のステートメントまたは局所変数を表示するには、代わりにプログラムウィンドウを使います。

ソース欄のコードは、CodeWarrior IDE エディタの設定パネルで指定したフォントとカラーで表示されます。ファイル欄で選択した項目がソースコードを含まない場合は、ソース欄に『ソーステキストは表示できません (Source text or disassembly not available)』という内容のメッセージが表示されます。

プログラムウィンドウと同様に、ブラウザウィンドウにも一番下にソースポップアップメニューがあります (『[ソースコードをアセンブラで見る](#)』(p27))。『Assembler』を選択すると、ソース欄の内容がアセンブラコードで表示されます (図 3.7)。アセンブラコード中にもソースコードと同様にブレークポイントを設定できます。『Mixed』を選択すると、ソースコードとアセンブラコードの両方が表示されます (図 3.8)。

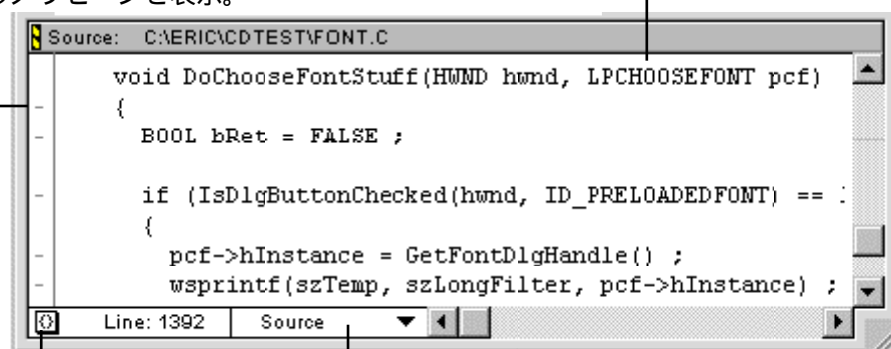
『[ソース欄](#)』(p25)、『[フォントや色を変更する](#)』(p63) と『[ブレークポイント](#)』(p63) もご覧ください。

図 3.14 ブラウザのソース欄

ファイルウィンドウのソース欄には、ライブラリが選択されているとき『Source text or disassembly not available』(ソーステキストや逆アセンブルは表示不可) というメッセージを表示。

ブレークポイント列でターゲットプログラムの実行を停止するブレークポイントを設定する。

このアイコンをクリックすると、ファイル欄で選択したファイルで定義されている関数の一覧を表示する。



このメニューをクリックして、ソースコード表示とアセンブラコード表示を切り替える。

### 関数ポップアップメニュー

ソース欄の左下にある関数ポップアップメニューには、ファイル欄で選択したソースファイルで定義されている関数のリストが入っています。関数欄で関数をクリックしたときと同じように、関数ポップアップメニューで関数を選択すると、その関数がソース欄に表示されます。

Alt/Option キーを押しながら関数ポップアップメニューをクリックすると、図 3.9 (p29) のようにアルファベット順にソートして表示します。

注意： ソース欄にソースコードが表示されていない場合、関数ポップアップメニューは何も表示しません。

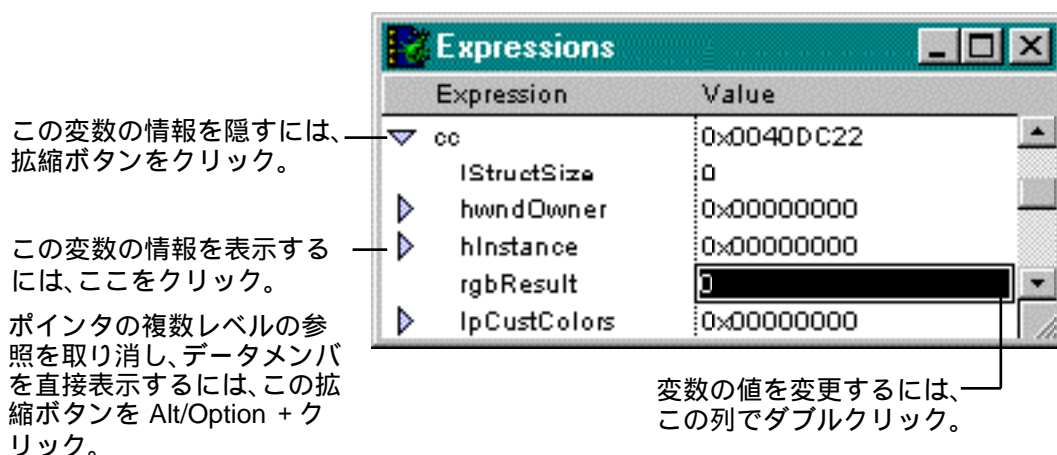
## Expression ウィンドウ

Expression ウィンドウ ( [図 3.15](#) ) は、頻繁に使用する局所変数や大域変数、構造体のメンバ変数や配列の要素を置いておくために使います。

Expression ウィンドウを開くには、Window メニューの『Expressions Window』コマンドを選択します。

Expression ウィンドウに選択した項目を追加するときは、Data メニューの『Copy to Expression』を使います。他の変数欄やウィンドウから Expression ウィンドウへ、マウスでドラッグすることもできます。また Expression ウィンドウのリスト内でドラッグして順序を変えることもできます。

図 3.15 Expression ウィンドウ



Expression ウィンドウから項目を削除するときは、項目を選択してから Backspace/Delete キーを押すか、Edit メニューで『Clear』を選択してください。

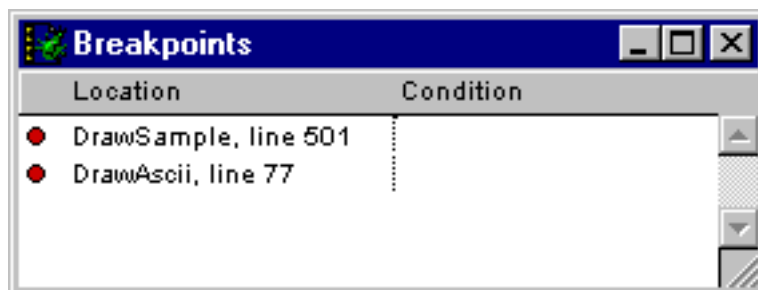
普通の変数ウィンドウと違って、Expression ウィンドウにある局所変数は、関数が呼び出し元に戻るときに、削除されません。

[『Show/Hide Expressions』\( p118 \)](#)、[『Copy to Expression』\( p113 \)](#) および [『Expression ウィンドウを使う』\( p78 \)](#) もご覧ください。

## Breakpoint ウィンドウ

Breakpoint ウィンドウ ( [図 3.16](#) ) は、カレントターゲットのブレークポイントの一覧を、ソースファイルと行番号で表示します。Breakpoint ウィンドウを開くには、Window メニューから『Show Breakpoints』を選択します。

図 3.16 Breakpoint ウィンドウ



各リストの左側にブレークポイントのマークが表示されます。丸いマークは、ブレークポイントがアクティブ状態にあることを示しています。横線のマークは、ブレークポイントがアクティブではない状態にあることを示しています。ブレークポイントウィンドウ上でブレークポイントマークをクリックして、ターゲットプログラムのブレークポイントをオン/オフに設定します。オフの状態でも位置は保持されます。ブレークポイントが設定された項目をダブルクリックすると、ブラウザウィンドウがアクティブになり、ソースコードがソース欄に表示されます。

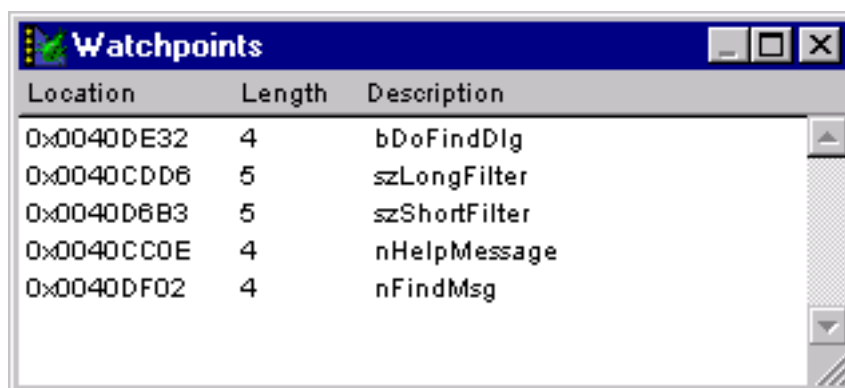
さらに、各ブレークポイントに条件（および評価式）を設定することもできます。評価式が TRUE でブレークポイントがオンの場合、そのブレークポイントでプログラムの実行が停止します。ブレークポイントがオフの場合、または評価式が FALSE の場合、ブレークポイントは何もしません。

[『ブレークポイント』\(p63\)](#)、[『Show/Hide Breakpoints』\(p118\)](#) および [『条件ブレークポイント』\(p66\)](#) もご覧ください。

## Watchpoint ウィンドウ

Watchpoints ウィンドウ ([図 3.17](#)) はプロジェクトで設定されたすべてのウォッチポイントをメモリアドレスで表示します。Watchpoints ウィンドウを開くには、Window メニューから『Show Watchpoints』を選択します。

図 3.17 Watchpoint ウィンドウ



ウォッチポイントを消去するには、マウスで選択して、次のいずれかを行います。

Data メニューから『Clear Watchpoint』を選択する。

Edit メニューから『Clear』を選択する。

Backspace/Delete キーを押す。

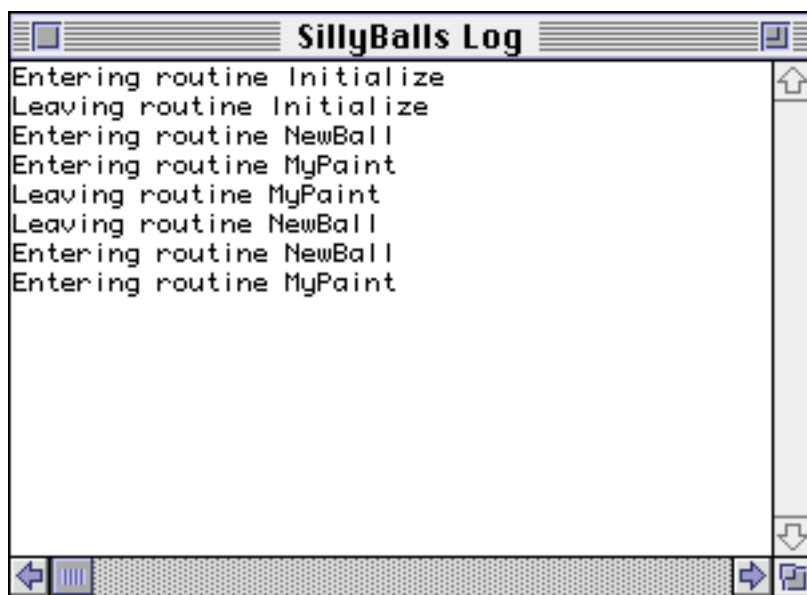
[『ウォッチポイント』\(p68\)](#) および [『Show/Hide Watchpoints』\(p118\)](#) もご覧ください。

## Log ウィンドウ

Log ウィンドウ ([図 3.18](#)) は、プログラムがシステム DLL または新しいタスクへの呼び出しを行ったときにメッセージを表示します。

Log ウィンドウの内容を直接編集できます。プログラムの実行に注釈を加えることもできます。Edit メニューの『Copy』コマンドを使ってメッセージをコピーしたり、File メニューの『Save』または『Save As』コマンドを使って、メッセージをテキストファイルとして保存し、後で解析のために使うことができます。

図 3.18 Log ウィンドウ



## 変数ウィンドウ

変数ウィンドウ ([図 3.19](#)) には、一つの変数が表示されます。このウィンドウを使って、変数の値を編集できます。局所変数を表示している変数ウィンドウは、変数を定義している関数が呼び出し元に戻るときに閉じます。

図 3.19 変数ウィンドウ



## 配列ウィンドウ

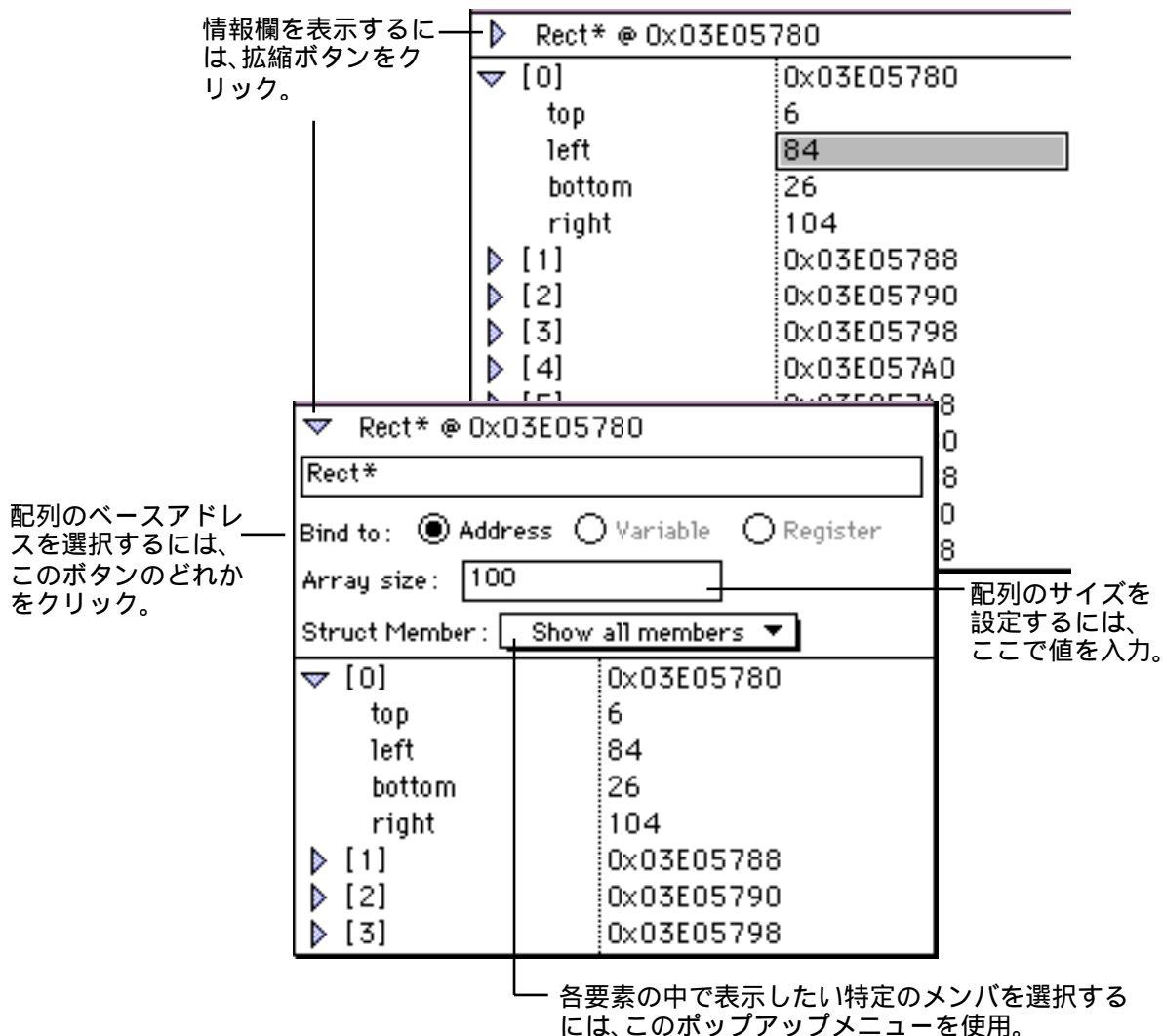
配列ウィンドウ (図 3.20) は、隣接しているメモリのブロックを、要素の配列として表示します。また、その要素の内容を編集することもできます。配列ウィンドウを開くには、局所変数欄または大域変数欄で配列変数を選択して、Data メニューの『View Array』を選んでください。配列ウィンドウを閉じるには、ウィンドウのクローズボックスをクリックしてください。

配列ウィンドウを開くために、Data メニューの『View Memory as』も使用できます。このコマンドはデータ型を選択するためのダイアログを開き、そこで選択された型の配列としてメモリを解釈して配列ウィンドウを開きます。

配列ウィンドウのタイトルバーは、配列の先頭アドレスを示します。配列の先頭アドレスは、アドレス、変数、またはレジスタに結び付けることができます。変数欄またはレジスタ欄からレジスタ名または変数名を配列ウィンドウにドラッグすることで、配列のアドレスを設定します。局所変数に結び付けられた配列は、変数を定義している関数が呼び出し元に戻るときに閉じられます。

情報欄には、配列の要素のデータ型と配列の先頭アドレスが表示されます。情報欄の矢印をクリックすると、配列の詳細が表示されます。拡張された情報欄には、配列の先頭アドレス、サイズ、および配列要素が構造型の一部である場合は、そのメンバが表示されます。

図 3.20 配列ウィンドウの詳細



配列の内容は、要素 0 から順にリストされます。配列の要素が構造体の形をとっている場合、各配列の要素の左側に拡張ボタンが表示されます。このボタンをクリックして、配列要素のメンバ変数を表示したり、隠したりすることができます。

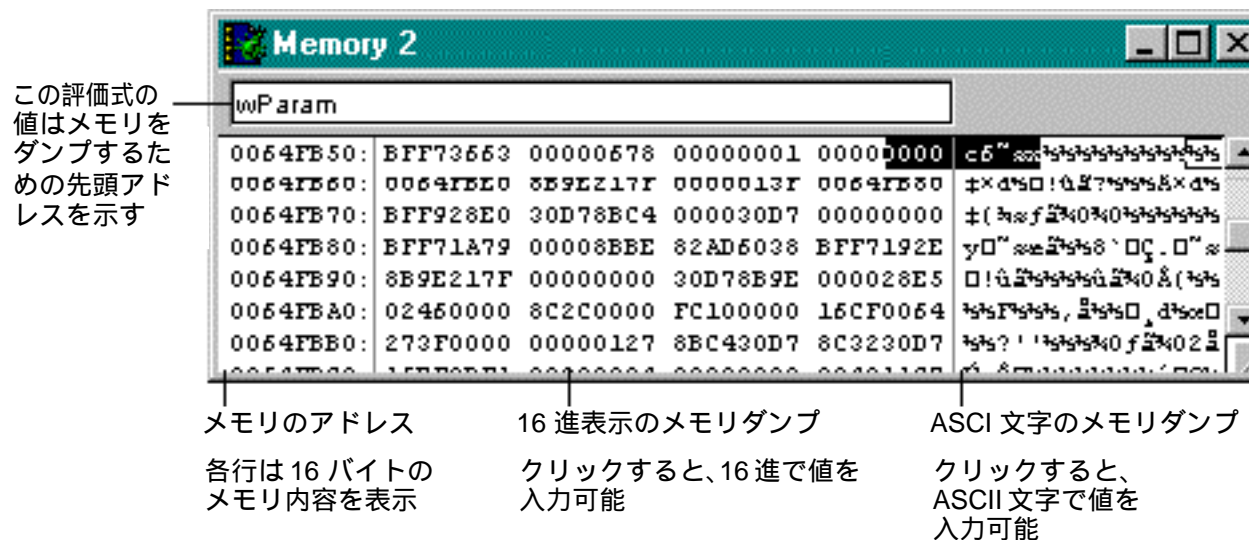
[『Open Array Window』\(p113\)](#) および [『View Memory As』\(p115\)](#) を参照してください。

## メモリウィンドウ

メモリウィンドウ ([図 3.21](#)) は、ローメモリの値を 16 進数と ASCII 文字で表示します。メモリウィンドウを開くには、プログラムウィンドウ、ブラウザウィンドウ、または Expression ウィンドウで、変数、関数名、または表示したい先頭アドレスを表す評価式を選択してから Data メニューの『View Memory』を選択してください。メモリウィンドウを閉じるには、クローズボックスをクリックします。



図 3.21 メモリウィンドウ



注意： メモリを特定の型のデータとして表示する配列ウィンドウ ([『配列ウィンドウ』\(p38\)](#)) を開くには、『View Memory as』を選択します。

先頭アドレスを変えるには、新しい評価式を式フィールドにタイプ入力するかドラッグします。評価式が、lvalue (左辺式) を与えないとき、評価式の値が先頭アドレスとなります。例えば、メモリウィンドウの評価式

PlayerRecord

は PlayerRecord のアドレスから始まるメモリ内容をダンプします。

評価式の結果がメモリ上のオブジェクト (lvalue) のとき、そのオブジェクトのアドレスを先頭としてメモリ内容がダンプされます。例えば、メモリウィンドウの評価式

\*myArrayPtr

は myArrayPtr が指しているオブジェクトのアドレスから始まるメモリ内容をダンプします。

メモリウィンドウで、メモリ内の値を変更することもできます。表示データ内で変更したいメモリの開始点をクリックして、タイプ入力します。16 進表示のデータを選択したときは、16 進数で入力します。ASCII 表示のデータを選択したときは、英数字で入力します。入力するとき、Backspace/Delete、Tab、Enter などのキーは使用できません。データを入力するとメモリ内の値は上書きされます。

Mac OS

評価式がポインタサイズのレジスタのとき、レジスタの値が先頭アドレスとなります。例えば、



®A0

は、68K レジスタ A0 の値のアドレスから始まるメモリを表示します。浮動小数点レジスタはこの用途には使えません。

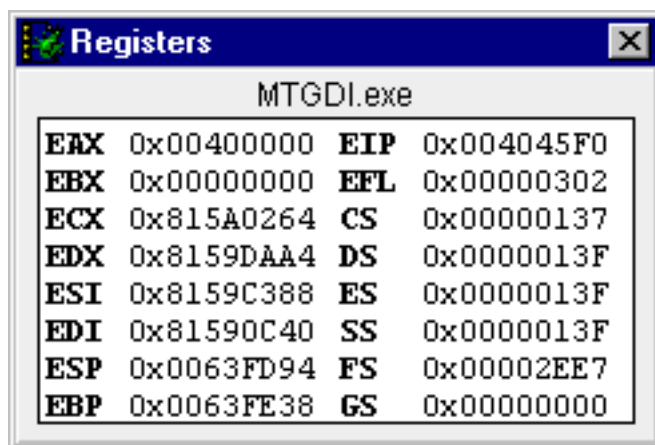
**警告！** メモリの内容を勝手に変更することは、非常に危険です。クラッシュを起こす恐れがあります。完全に把握していない場合、データを絶対に変更しないでください。

## レジスタウィンドウ

レジスタウィンドウには CPU レジスタ ( [図 3.22](#) ) が表示され、このレジスタの内容を編集できます。レジスタウィンドウを開くには、Window メニューのサブメニューにある『General Registers』を選択します。

注意： ターゲットプロセッサによってレジスタウィンドウの外観は変わります。Window メニューのサブメニューがない場合、『Register Window』を選択します ( [図 3.22](#) )。

図 3.22 CPU レジスタウィンドウ

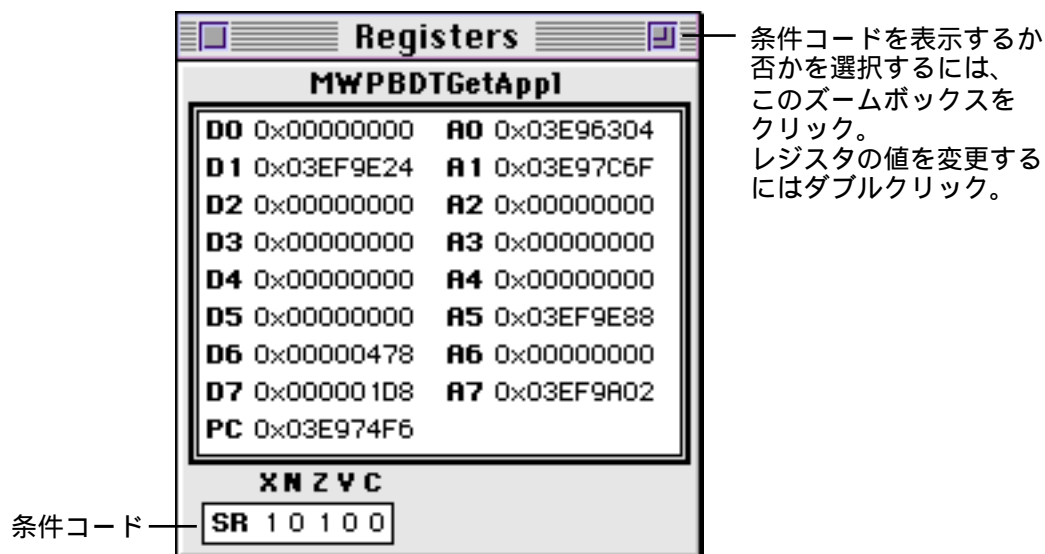


レジスタの値を変更するには、その値をダブルクリックする。

ターゲットによっては FPU ( Floating-Point Unit ) レジスタを利用できます。この場合、Window メニューのサブメニューの『FPU Registers』を選択すると FPU レジスタを表示できます。

レジスタの値を変更するには、レジスタ値をダブルクリックするか、またはレジスタを選択してから Enter/Return キーを押します。この後、新しい値を入力します。

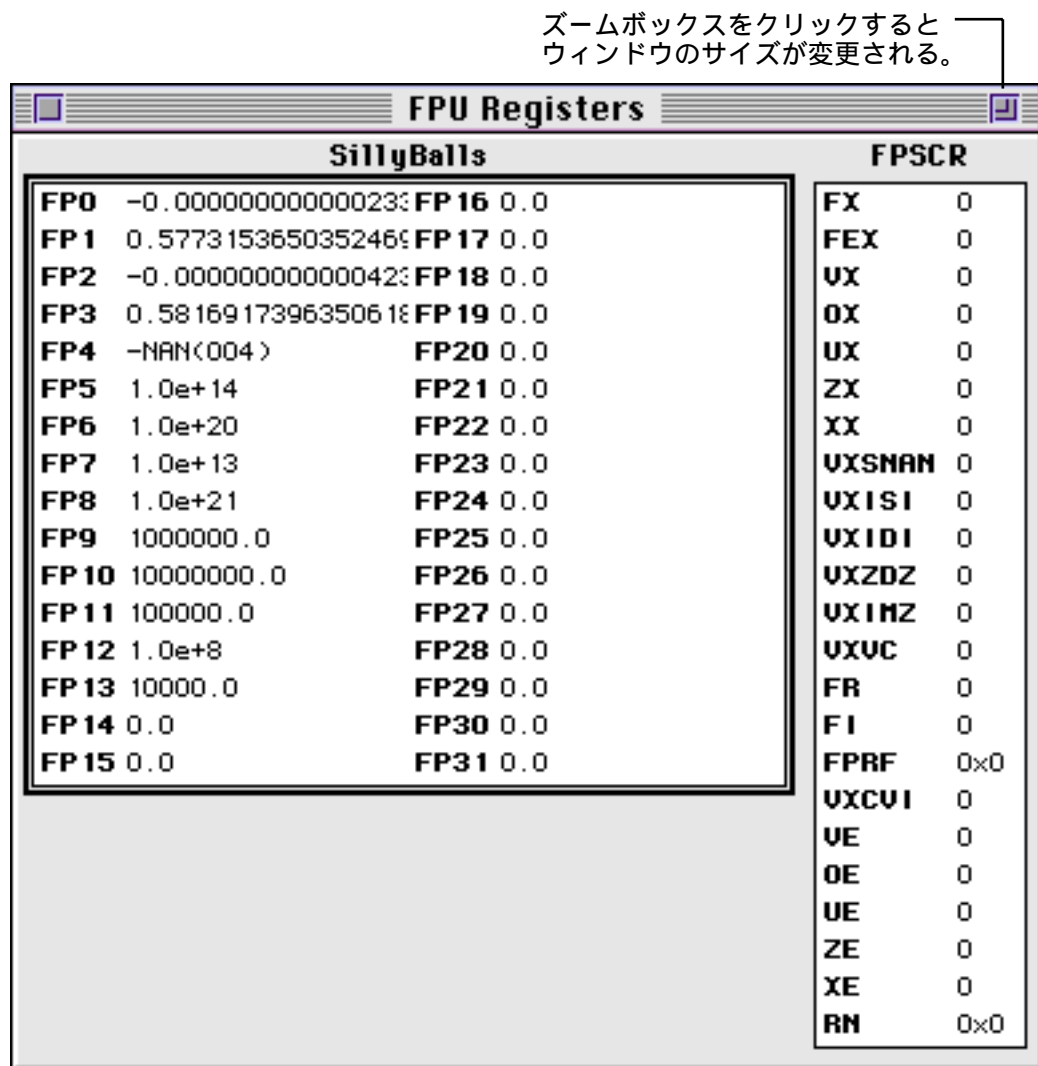
図 3.23 CPU レジスタウィンドウ (Mac OS)



Mac OS : レジスタのズームボックスをクリックするとすべてのレジスタが表示されます。  
ステータスレジスタや条件レジスタを 0 と 1 の間でトグルするには、レジスタをダブルク  
リックするか、レジスタを選択してから Return キーか Enter キーを入力してください。

**警告!** レジスタの値を変えることは非常に危険です。データやメモリを壊して、クラッ  
シュを引き起こすこともあります。

図 3.24 FPU レジスタウィンドウ (Mac OS)

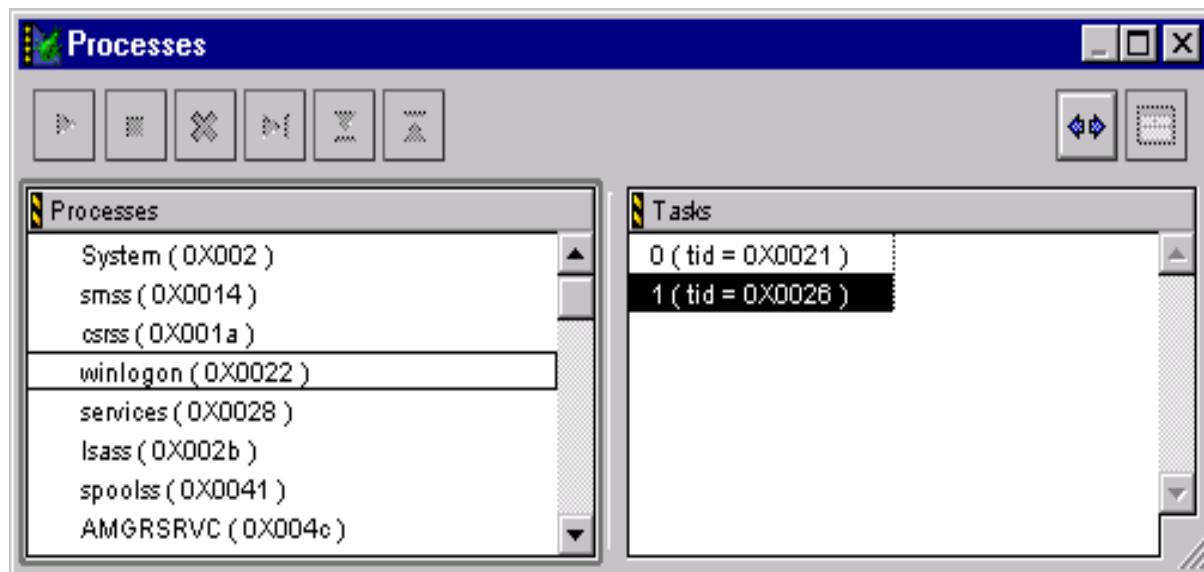


[『Show/Hide Registers』 \( p118 \)](#) および [『Show/HideFPU Registers』 \( p118 \)](#) もご覧ください。

## プロセスウィンドウ

プロセスウィンドウ ( 図 3.25 ) には現在実行中のプロセスが、隠しプロセスも含めて表示されます。プロセスウィンドウには、選択したプロセスのタスクも表示されます。プロセスウィンドウを開くには、Window メニューの『Processes Window』を選択します。

図 3.25 プロセスウィンドウ



プロセスウィンドウには、次の三つの欄があります。

[プロセス欄](#)：現在実行中の全プロセス名を表示します。

[タスク欄](#)：選択したプロセスにおける全タスクを表示します。

[プロセスウィンドウのツールバー](#)：プロセスやタスクをデバッガのコントロール下で実行、終了、停止します。

[『Show/Hide Processes』\(p117\)](#) もご覧ください。

## プロセス欄

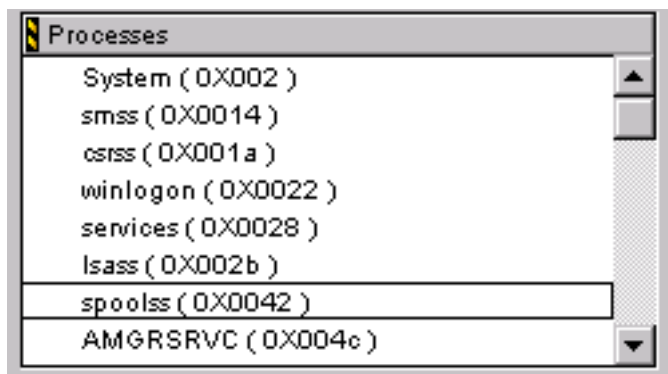
プロセス欄はアクティブなすべてのプロセスを表示します。デバッガのコントロール下にあるプロセスにはチェックマークが付いています。デバッガのコントロールを設定するには、チェックマーク列をクリックしてください。プロセス名をダブルクリックするとプロセスをアクティブにします。

---

ヒント： プロセスのデバッガコントロールをオンに設定しているとき、そのプロセスを終了することなく、非ターゲット化することができます。プロセスを非ターゲット化するには、該当するチェックマーク列をクリックし、続いて表示されるダイアログで [ Resume ] をクリックします。

---

図 3.26 プロセス欄

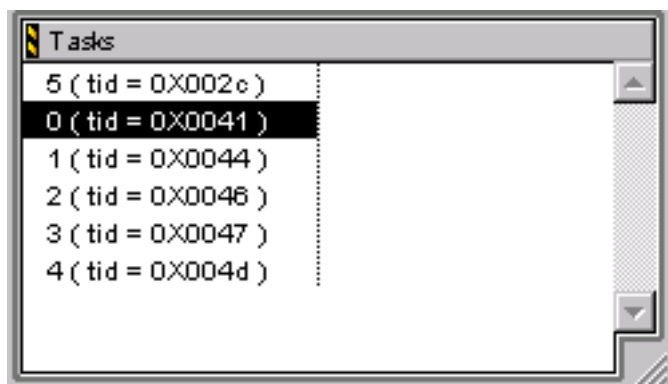


### タスク欄

タスク欄は指定したプロセスに所属するアクティブなタスクをすべて表示します。デバッガコントロール下のプログラムのタスクのみが表示されます。タスク名をダブルクリックすると、そのタスクのコードを含むプログラムウィンドウがアクティブになります。タスクを選択して右上にあるプログラムウィンドウのボタンをクリックしても同様のことができます。

タスク欄には二つの列があります。最初の列はタスク ID を示し、次の列はタスク状態を示します。タスク状態は実行、停止、クラッシュのいずれかです。

図 3.27 タスク欄

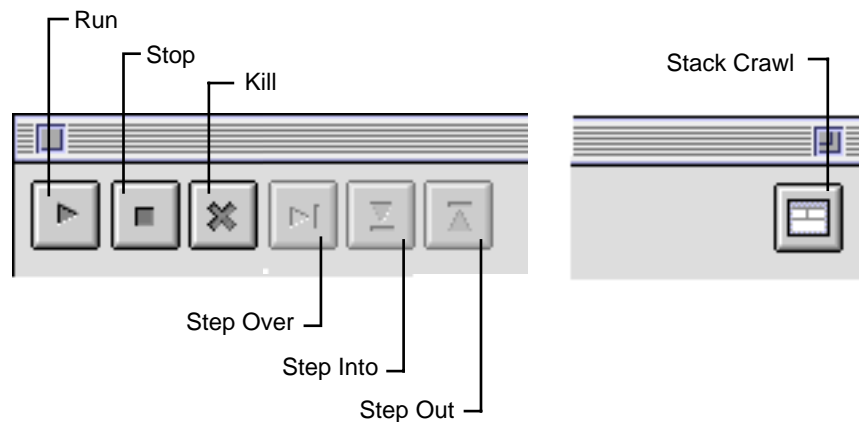


### プロセスウィンドウのツールバー

プロセスウィンドウのツールバー (図 3.28) には、デバッガコントロール下のプロセスを実行 (Run)、停止 (Stop)、終了 (Kill) するボタンがあります。これらの機能は、他のアクティブなプロセスおよびタスクには影響を与えません。

[ Step Over ], [ Step Into ], [ Step Out ] ボタンの動作は、プログラムウィンドウのボタンの動作と同じです。これらのボタンをクリックすると、プログラムウィンドウを表示し、カレントステートメント矢印を示します。

図 3.28 プロセスウィンドウのツールバー



プログラムウィンドウのボタンは、選択されているプロセスまたはタスクのプログラムウィンドウを表示します。プロセスが選択されると、そのプロセスのプログラムウィンドウが前に出ます。タスクが選択されると、そのタスクのプログラムウィンドウが前に出ます。同時に複数のプログラムウィンドウを表示できます。

[『デバッガのツールバー』\(p24\)](#) もご覧ください。



## 第 4 章 デバッガの使い方

この章では、デバッグの基本を解説します。

### デバッガの概要

デバッガは、プログラムの実行をコントロールするアプリケーションです。デバッガを使って処理の流れを見たり、トラブル箇所を見つけたりすることができます。この章では、プログラムの実行をコントロールしながら、データや変数の値を見たり、変更したりすることで、ソースコードの問題箇所を見つけて解決するための CodeWarrior デバッガの使い方について説明します。この章で扱う内容を以下に示します。

[デバッガを起動する](#)：デバッガを起動したときに見えるウィンドウについて説明します。

[コードの実行、ステップ実行、停止](#)：プログラムを一行ずつ実行する方法を説明します。

[ナビゲーション](#)：プログラムの実行をコントロールする方法を詳しく説明します。

[ブレークポイント](#)：実行を停止する方法について説明します。

[ウォッチポイント](#)：注目したいメモリ位置の内容が変更されたとき実行を停止する方法を説明します。

[データの表示、変更](#)：変数を見たり、変更する方法について説明します。

[ソースコードを編集](#)：デバッグ中にソースコードを編集する方法を説明します。

デバッガを使うためにターゲットを準備する方法については、『[この章の概要](#)』(p15)をご覧ください。この章は『[ウィンドウの概要](#)』(p21)をご覧ください。既にデバッガのツールバーやウィンドウについての知識がある方を対象としています。デバッガの環境設定については、『[Preferences](#)』(p110)をご覧ください。

### デバッガを起動する

統合された CodeWarrior デバッガを起動するには、プロジェクトを開いてから Project メニューの『Enable Debugger』を選択してください。次に Project メニューの『Debug』を選択するとデバッガが起動します。

MW Debug を使う場合、以下の点に注意してください。

MW Debug が起動されていない場合、プロジェクトから起動するか、またはシンボルファイルから直接起動すると、デバッガはプログラムウィンドウをアクティブウィンドウにして起動します。この場合は正常に動作しています。プログラムコードはプログラムウィンドウに表示され、最初の行で停止しており、実行の準備が整っています。

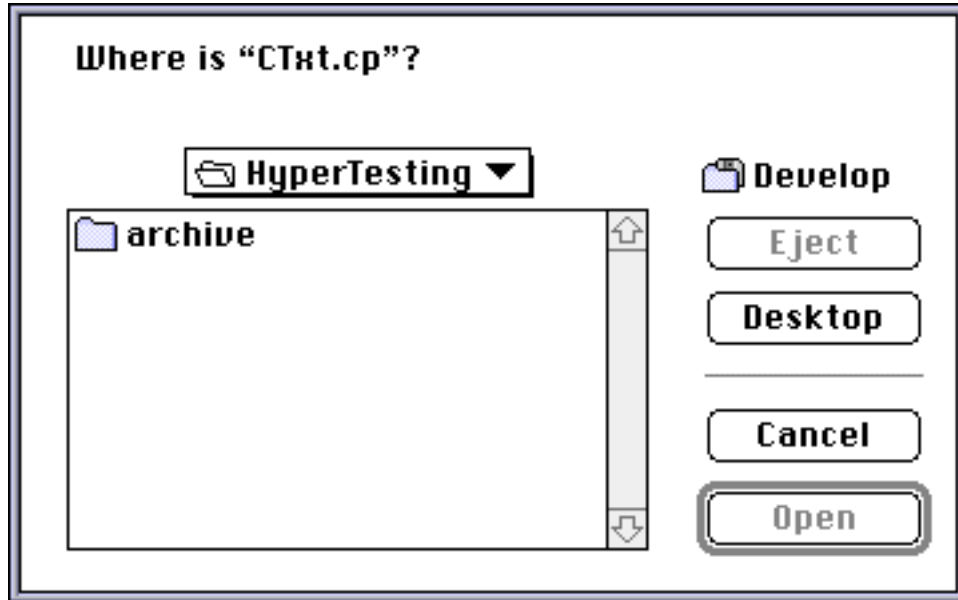
MW Debug が既に起動されている場合、プロジェクトから直接起動すると、デバッガはブラウザウィンドウをアクティブウィンドウにします。この場合はプロジェクトを実行するためにデバッガの『Run』コマンドを選択します。このコマンドにより、ターゲットがデバッガのコントロール下で起動し、プログラムウィンドウがアクティブになり、プログラムの最初の行で停止します。

図 4.1 ファイルの位置を尋ねる





図 4.2 ファイルの位置を尋ねる (Mac OS)



デバッガが該当ファイルのある位置を尋ねるダイアログ (図 4.1) を表示することがあります。

このダイアログが表示されるのはデバッガを起動したとき (デバッガがメインエントリーポイントを持つファイルを検索するとき) または MW Debug の [ブラウザウィンドウ](#) であるファイルをクリックしたときです。

以下の状況でもダイアログが表示されます。

ファイルが他のディレクトリに移動されているとき。

他人から受け取ったプロジェクトファイルのパスが異なるとき。

コンパイルされたライブラリに属するファイルを選択し、かつそのソースファイルを持っていないとき。

デバッガシンボルをオフにしてコンパイルされたライブラリをターゲットで使用している場合、最後のケースがもっともありがちなものです。CodeWarrior にはコンパイルされたライブラリが含まれていますので特にそうなることがあります。

一度ファイルを指定すると、デバッガはその位置を (デバッグ中であっても) 記憶します。

[『MW Debug を IDE から起動 \(Mac OS\)』\(p18\)](#) および [『MW Debug を直接起動』\(p19\)](#) もご覧ください。







## コードの実行、ステップ実行、停止

ここでは、プログラム実行のコントロール方法について説明します。これは一行ずつの実行、さらにデバッグが終わるときのターゲットの停止または終了の方法を含みます。

一行ずつコードを実行することを『ウォークスルー（walk through）』またはステップ実行と呼ぶこともあります。これは、プログラムの最初から始めて、コードを順にたどっていく地道な方法であり、コードの流れを理解するには重要です。しかし、デバッガの本当の力を発揮するには、次の節で紹介するような高度な手法を必要とします。この手法は、任意の位置へ直接ナビゲートしたり、特定の位置で特定の条件を満たしたときにコードを停止したり、さらにデータの表示や変更する方法を含みます。

コードのウォークスルーには、いくつかの方法があります。ツールバーのボタン、キーボード、またはデバッガの Control メニューのコマンドを選択することにより、コードを進めます。[表 4.1](#) に、ボタンと等価なキーボード操作を示します。

表 4.1 ボタンとキー操作によるコマンド

ボタン	メニュー コマンド	キーボード操作 ( Windows )	キーボード操作 ( Mac OS )
	Run	F5	Command + R
	Stop		Control + P
	Kill	Shift + F5	Control + K
	Step Over	F10	Control + S
	Step Into	F11	Control + T
	Step Out	Shift + F11	Control + U

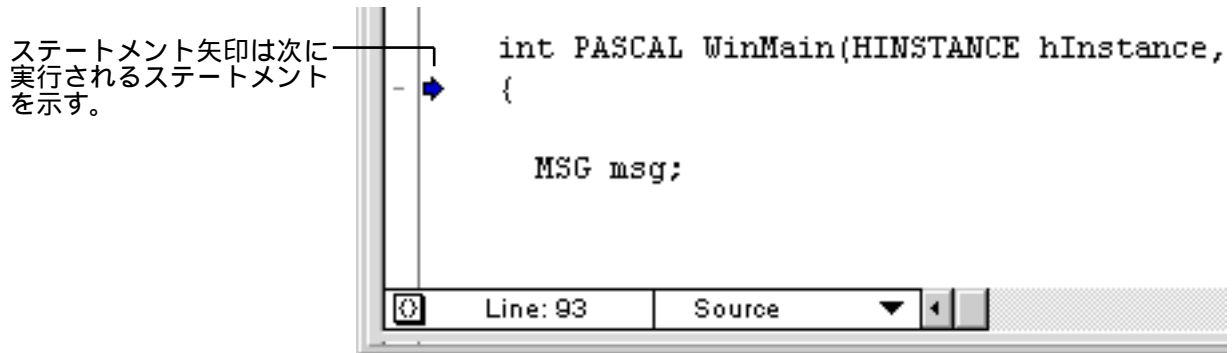
ここでは以下の内容について説明します。

- [カレントステートメント矢印](#)
- [コードを実行](#)
- [一行ずつ実行する](#)
- [ルーチンの中へ入る](#)
- [ルーチンの中から呼び出し元に戻る](#)
- [ステートメントをスキップする](#)
- [実行を停止する](#)
- [プログラムを終了する](#)

## カレントステートメント矢印

プログラムウィンドウのカレントステートメント矢印 (図 4.3) は、次に実行されるステートメントを指します。これは、プロセッサのプログラムカウンタレジスタが指している位置と同じです。デバッガを起動した直後、カレントステートメント矢印はプログラムの実行コードの最初の行を指しています。

図 4.3 カレントステートメント矢印



## コードを実行

ターゲットを起動して実行が停止している状態でプログラムを実行するには、Control メニューの『Run』を選択するか、ツールバーの [ Run ] ボタンをクリックしてください (図 4.4)。カレントステートメント行からプログラムが実行されます。

図 4.4 『Run』コマンド



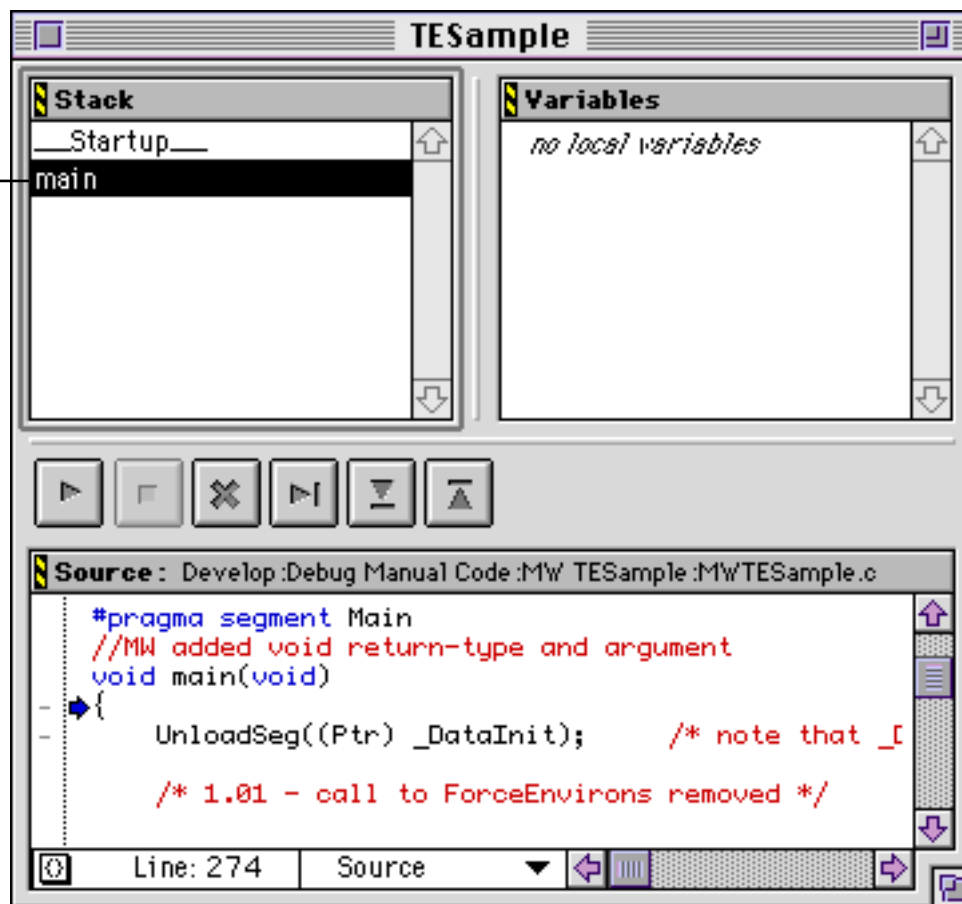
注意： ( MW Debug ) ターゲットが実行されると、プログラムウィンドウのソース欄にソースコードが表示されます。プログラムウィンドウに『Program name is not running.』 (プログラムが実行中ではありません) というメッセージが表示されたときはターゲットを実行できません。このような場合は、デバッガで『Run』コマンドを選択してください。プログラムウィンドウが最前面に表示され、最初のコードで実行が停止します。

ブレークポイントまたは『Stop』で停止すると、コントロールはデバッガに戻り、カレントステートメントと、局所変数および大域変数の現在の値が表示されます。デバッガは、暗

默のブレークポイントをプログラムのメインエントリーポイントに置いて、そこで停止します(図 4.5)。ここで『Run』コマンドを使うと、割込みの起きた場所からプログラム実行が再開されます。『Kill』コマンドの後の『Run』コマンドは、プログラムを最初から再実行します。

図 4.5 ターゲットプログラムの実行を開始する

プログラムがデバッガから起動されたとき、プログラムの実行は自動的に main のエントリーポイントで停止する。



ヒント： MW Debug の自動起動を禁止するには、Alt/Option キーを押しながらシンボルファイルを開きます。『[Automatically launch applications when SYM file opened](#)』オプションを変えておくこともできます(『[Program Control 環境設定パネル](#)』(p99)を参照)。この機能は、プログラムのメインルーチンの前に実行される、C++ の static コンストラクタ関数のデバッグなどに有効です。

### 一行ずつ実行する

ステートメントを一行ずつ実行するときは、『Step Over』コマンドを選択します(図 4.6)。ステートメントがルーチンコールの場合、ルーチン全体が実行され、カレントステートメ

ント矢印はコードの次の行に進みます。呼び出されたルーチンの内容も一行ずつ実行されますが、デバッガのプログラムウィンドウには表示されません。つまり、『Step Over』コマンドは、呼び出されたルーチンのコード内容にはふれずにルーチンコールを実行します。コードを一行ずつ実行してルーチンの終わりに達すると、カレントステートメント矢印はルーチンの呼び出し元に戻ります。

図 4.6 『Step Over』コマンド



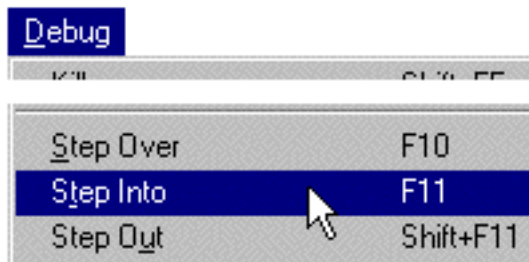
次のステートメントを実行するには、Control メニューの『Step Over』を選択するか、ツールバーのこのアイコンをクリックする。これはルーチンコールの中には入らない。



## ルーチンの中へ入る

呼び出したルーチンの実行を追いかけてみたい場合は、コードをトレースします。ルーチンコールにステップインし、ステートメントを一行ずつ実行するときは、『Step Into』コマンドを選択します (図 4.7)。

図 4.7 『Step Into』コマンド



ターゲットプログラムの関数の中へ入るには、Control メニューの『Step Into』を選択するか、ツールバーのこのアイコンをクリックする。

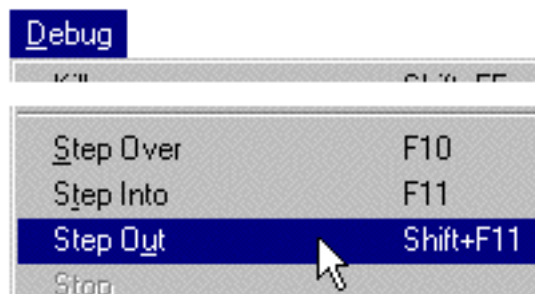


『Step Into』は、現在のステートメントがルーチンコールでなければ、カレントステートメントを 1 ステートメント下に移動します。『Step Into』がルーチンコールにあたると、呼び出されたルーチンの中へ実行を移します。

## ルーチンの中から呼び出し元に戻る

現在のルーチンを実行した後に、呼び出し元に戻るときは、『Step Out』コマンドを選択します (図 4.8)。『Step Out』コマンドは、ステートメント矢印が指しているステートメントからプログラムを実行し、ルーチンがその呼び出し元に戻るときに、デバッガに戻ります。コールチェーンを 1 レベル『上に』戻ります。『[コールチェーンによるナビゲーション](#)』(p57) をご覧ください。

図 4.8 『Step Out』 コマンド



ターゲットプログラムの関数から呼び出し元へ戻るには、Controlメニューの『Step Out』を選択するか、ツールバーのこのアイコンをクリックする。



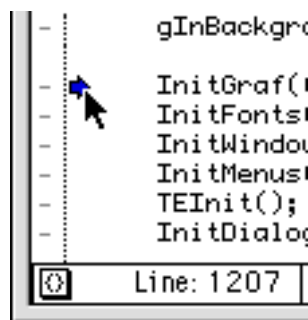
## ステートメントをスキップする

ソースコードの別の位置へ、コードを実行せずに移動したい場合があります。現在実行中のソース部分とは異なる位置へステートメント矢印を移動するには、ステートメント矢印をドラッグしてください(図 4.9)。ステートメント矢印をドラッグしても、ドラッグ前の位置からドラッグした位置までのコードは実行されません。

**警告!** ステートメント矢印をドラッグすると、レジスタウィンドウのプログラムカウンタを変更することになります。これはとても危険な操作です。ルーチンコールやルーチンからの戻りをスキップすることがスタックを破壊することにつながるからです。デバッガはこのような状況を救うことはできないので、この操作を行うときは十分な配慮が必要です。

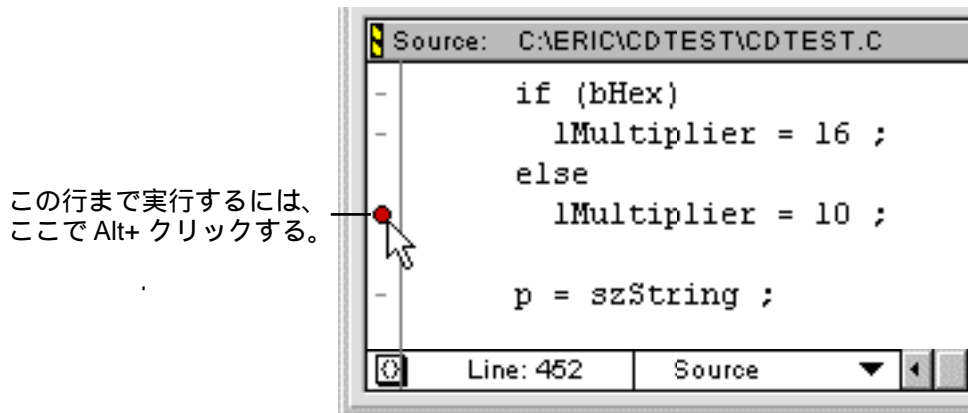
図 4.9 カレントステートメント矢印をドラッグする

カレント行を変更するには、ステートメント矢印をドラッグする。



危険を犯さずにステートメント矢印を移動するには、ブレークポイント列のステートメントを Alt/Option + クリックしてください(図 4.10)。その位置に一時的なブレークポイントが設定され、そこまで正常に実行されます(『ブレークポイント』(p63)をご覧ください)。

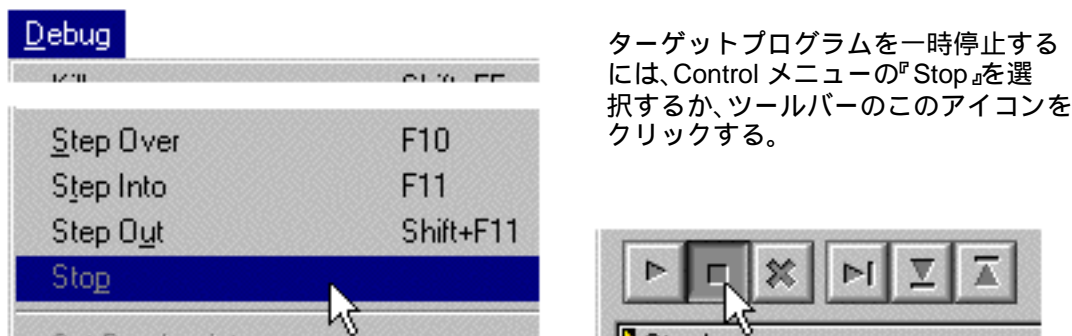
図 4.10 一時的なブレークポイントを設定する



### 実行を停止する

プログラムの実行を停止するには、Control メニューの『Stop』コマンドを選択してください(図 4.11)。これによってオペレーティングシステムからデバッガにコントロールが移り、ある場所で実行が停止します。この後、一行ずつ実行したり、『Run』コマンドで実行を再開することもできます。

図 4.11 『Stop』コマンド



『Stop』コマンドは、停止する場所を指定できません。コードの実行は高速であるため、『Stop』コマンドを使ったときにどこで止まるかは正確に指定できません。停止する場所を正確に指定したいときは、ブレークポイントを使います(『ブレークポイント』(p63)をご覧ください)。

注意：『Stop』コマンドはオペレーティングシステムに依存するため、ターゲットによっては使用できません。詳細は各『Targeting』マニュアルを参照してください。

ヒント：(Mac OS) 無限ループでプログラムがハングアップしたとき、Command + Control + / キーを入力するとデバッガへ戻ることができます。これはプログラムに割り込

みをかけ、デバッガへ戻ることを可能にしています。したがって、デバッガで問題箇所を修正することができます。

ヒント： (Windows) 無限ループでプログラムがハングアップしたとき、CodeWarrior デバッガに切り替えて、『Stop』コマンドを使うとコントロールを回復できます。

## プログラムを終了する

プログラムを完全に終了するには、Control メニューの『Kill』コマンドを選択するか、ツールバーの『Kill』アイコンをクリックしてください(図 4.12)。プログラムが終了します。プログラムウィンドウには、プログラムが実行されていないことを示すメッセージが表示されます。

図 4.12 『Kill』コマンド



ターゲットプログラムを終了するには、Control メニューの『Kill』を選択するか、ツールバーのこのアイコンをクリックする。



プログラムの終了は停止と異なります。『Stop』コマンドはプログラムを一時停止するだけで、停止した位置から再開できますが、『Kill』コマンドはプログラムを終了します。

## ナビゲーション

プログラムをナビゲーションする（渡り歩く）方法を説明します。これは、ブレークポイントを必要な箇所に設定するために必要です。ここでは、プログラムをナビゲーションする基本的な方法を説明します。

[一般的なナビゲーション](#)：コードをステップ実行します。

[コールチェーンによるナビゲーション](#)：アクティブなルーチンへ移動します。

[ブラウザウィンドウによるナビゲーション](#)：MW Debug のブラウザウィンドウのコードに移動します。

[ソースコードによるナビゲーション](#)：ソースファイルのコードに移動します。

[Find ダイアログの使い方](#)：特定の定義、変数、またはルーチンコールを検索します。



## 一般的なナビゲーション

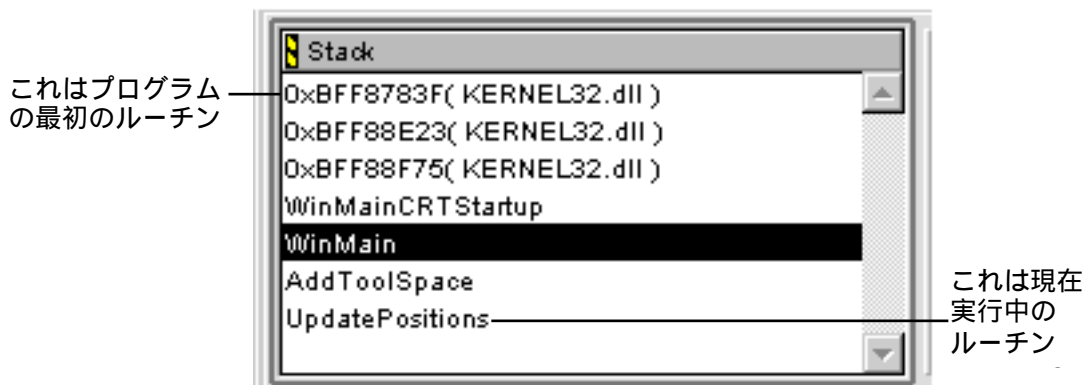
『Step Over』、『Step Into』、『Step Out』コマンドを選択してプログラムを実行しながら、必要な箇所に移動できます。コードを少しずつ追いたいときには有効な方法ですが、かなり離れたところにある特定のコードまで実行するには、有効な方法とはいえません。

[『一行ずつ実行する』\(p52\)](#) [『ルーチンの中へ入る』\(p53\)](#) および [『ルーチンの中から呼び出し元に戻る』\(p53\)](#) もご覧ください。

## コールチェーンによるナビゲーション

プログラムウィンドウのスタッククロール欄には、現在呼び出されているルーチンのチェーンが表示されます( [図 4.13](#) )。チェーン内のそれぞれのルーチンは、呼び出し元のルーチンの下に表示されます。現在実行中のルーチンは、チェーンの一番下にあり、プログラムで最初に実行されたルーチンは一番上にあります。

図 4.13 スタッククロール欄



スタッククロール欄を使って、現在実行中のルーチンを呼び出したルーチンにナビゲートできます。スタッククロール欄のルーチンを呼び出したルーチンを見つけるには、呼び出し元の名前をクリックします。呼び出し元のソースコードの呼び出した点が表示されます( [図 4.14](#) )。

図 4.14 ルーチン呼び出しの位置を見つける

DoMenuCommand() の  
呼び出し元の  
DoEvent() をクリック  
すると、DoEvent() で  
DoMenuCommand() を  
呼び出している部分を  
表示する。



## ブラウザウィンドウによるナビゲーション

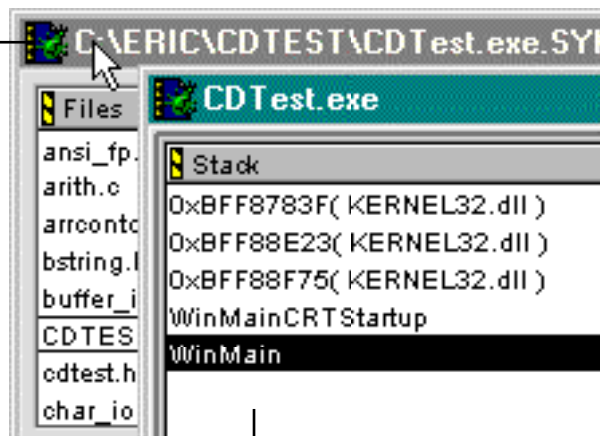
MW Debug のブラウザウィンドウを使って、ソースコード内の任意の位置にジャンプできます。特定のルーチンを表示するには、以下のようにします。

1. ブラウザウィンドウをアクティブにします ( 図 4.15 )。

図 4.15 ブラウザウィンドウをアクティブにする

アクティブにするには、  
ブラウザウィンドウを  
クリックする。

ブラウザウィンドウで  
ターゲットプロジェクト  
に含まれるすべての  
ファイルを見る。

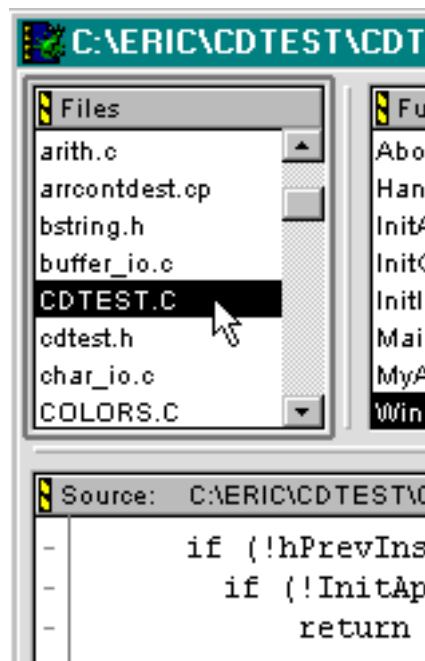


プログラムウィンドウ

2. ブラウザウィンドウのファイル欄で、見たいルーチンを定義しているファイルを選択します ( [図 4.16](#) )。

見たいファイルをクリックするか、矢印キーを使ってリストをスクロールしてください。そのファイルのソースコードがソース欄に表示されます。ファイル名を入力してもよいです。

図 4.16 内容を見たいファイルを選択する

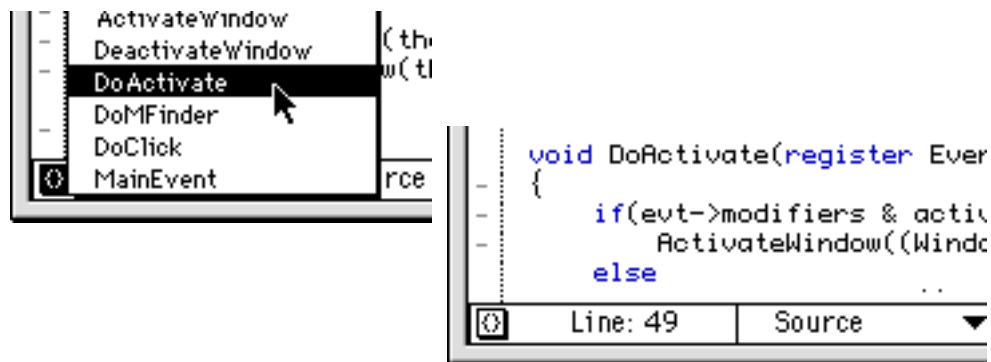


3. ソースファイル内で見たいコードを見つけます。

ソース欄をスクロールして望みのルーチンを見つけることができます。ブラウザウィンドウの関数欄や関数ポップアップメニュー ( [図 4.17](#) ) を使ってルーチンを選択する方が効率的です。

ブラウザウィンドウのソース欄にルーチンが表示されます。ルーチンが表示された後はソースを見たり、ブレークポイントを設定または消去できます (『[ブレークポイント](#)』[\(p63\)](#) をご覧ください)。

図 4.17 ルーチンを選択する



### ソースコードによるナビゲーション

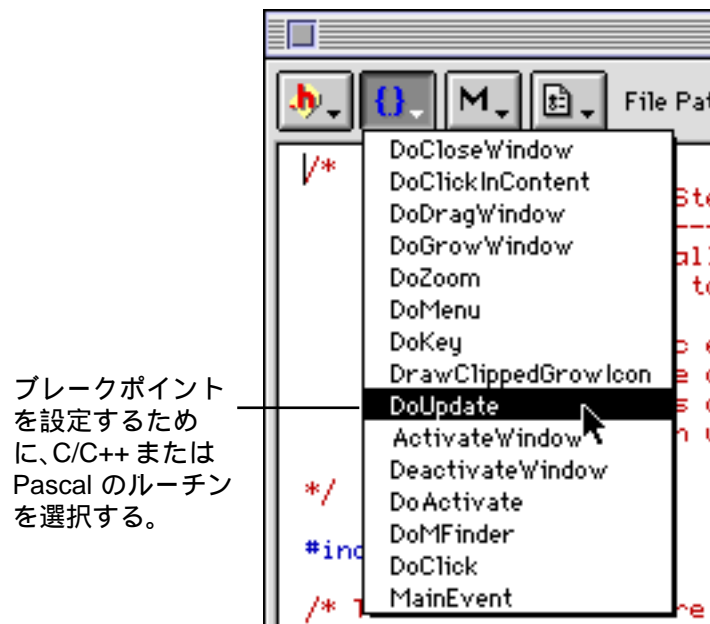
MW Debug のブラウザウィンドウにルーチンを表示したいときは、まず CodeWarrior IDE (C/C++ または Pascal いずれでも) でソースコードを開き、コードを検索します。その後、MW Debug に切り替えて同じコードをブラウザウィンドウに表示します。

CodeWarrior IDE を使って見たいルーチンまたはファイルを表示し、次に MW Debug のブラウザウィンドウにそのコードを表示する手順は以下の通りです。

1. CodeWarrior IDE で該当ルーチンを含むファイルを開きます。そのファイルはプロジェクトファイルでなければなりません。
2. ブラウザウィンドウで表示したい部分にカーソルで挿入位置を置きます。

見たいルーチンを表示するために、関数ポップアップメニュー (図 4.18) を使ったり、その他の方法を使って望みのコードを表示してください。

図 4.18 C/C++ または Pascal の関数を選択する

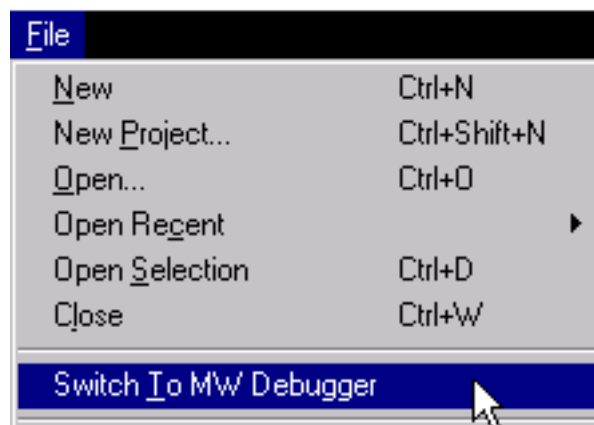


ヒント： Edit メニューの『Find...』または『Find Next』を使って、ブラウザウィンドウで該当コードを素早く表示することもできます。

3. File メニューの『Switch To MW Debugger』を選択します (図 4.19)。

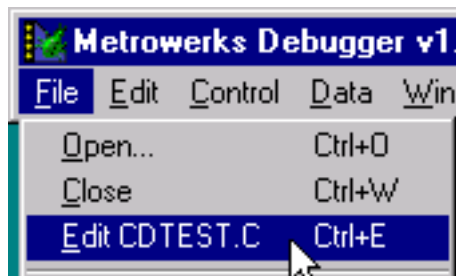
MW Debug がアクティブなアプリケーションとなります。ブラウザウィンドウは、IDE のエディタで置いたカーソルの挿入位置のステートメントを表示します。MW Debug の File メニューの『Edit Filename』を選択すると、いつでも IDE へ戻ることができます (図 4.20)。

図 4.19 MW Debug へ戻る



エディタのソース行をデバッガで表示するには、『Switch To MW Debugger』コマンドを選択する。

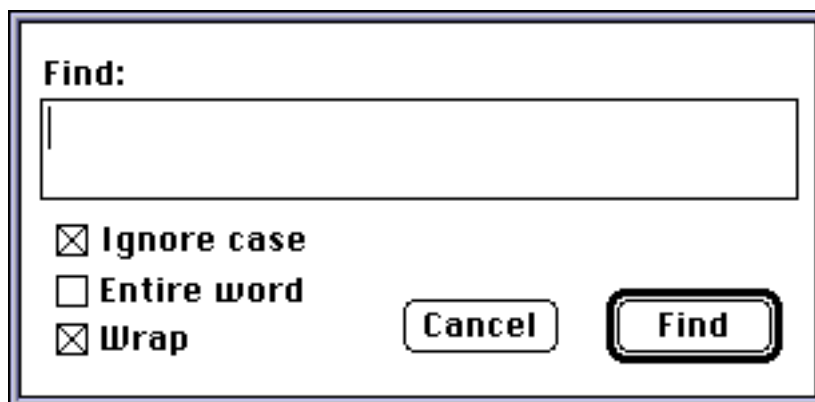
図 4.20 MW Debug から CodeWarrior IDE へ戻る



## Find ダイアログの使い方

MW Debug の『Find』ダイアログ (図 4.21) を使うと、プログラムウィンドウまたはブラウザウィンドウのソース欄でテキストを検索できます。検索は、選択の現在位置または挿入点から始まり、ファイルの終わりに向かって行われます。Edit メニューの『Find』を選択します。

図 4.21 MW Debug の Find ダイアログ



MW Debug の『Find』ダイアログには以下の項目が含まれます。

Text : 検索するテキストを入力するテキストフィールドです。

Ignore case : これをオンにすると、検索の際に大文字と小文字を区別しません。大文字と対応する小文字 (A と a など) を同一であると見なします。オフにすると、検索は大文字と小文字を区別して行われます。大文字と小文字は別の文字と見なされません。

Entire word : これをオンにすると、検索文字列と同じ完全な語 (区切り記号または空白文字で区切られた語) を検索します。オフにすると、検索文字列が他の長い語に含まれているような場合も検索対象となります。

Wrap : これをオンにすると、検索がファイルの終わりに達したときに、ファイルの先頭から続けます。オフにすると、検索はファイルの終わりに達した時点で終わります。

Find : ダイアログの内容を確認して、検索を始めます。ダイアログ内の設定は記憶され、次回『Find』コマンドが使われるときに再度表示されます。

Cancel : 検索を行わないでダイアログを閉じます。ダイアログ内の設定は記憶されず、次回『Find』コマンドが使われるときに表示される内容は以前記憶されたものになります。

『Find Next』コマンドを使って、選択の現在位置または挿入点から最後の検索を繰り返すことができます。

『Find Selection』コマンドを使って、ソース欄で現在選択されているテキストが次に現れる箇所を検索できます。このコマンドは、選択されているテキストがない場合や、挿入点だけの場合は使えません。

---

ヒント : Shift キーを押しながら、Ctrl/Command + G ( Find Next ) または Ctrl/Command + H ( Find Selection ) を使うと、反対方向に検索できます。

---

## フォントや色を変更する

デバッガのソースコードのフォントおよび色は、CodeWarrior IDE の Edit 設定パネルで指定します。

フォントやシンタックスカラーを変更する手順は以下の通りです。

1. CodeWarrior IDE を起動します。
2. エディタウィンドウおよびプロジェクトウィンドウが開いていないことを確認してください。
3. Edit メニューの『Preferences』を選択します。
4. Editor 設定パネルを選択します。
5. フォントおよびシンタックスカラーを設定します。
6. 設定パネルの [ OK ] ボタンをクリックします。

デバッガはこのフォントやシンタックスカラーの設定を使って、ソースコードを表示します。

## ブレイクポイント

ブレイクポイントは、ターゲットプログラムの実行を一時停止し、コントロールをデバッガに戻します。デバッガは、ブレイクポイントが設定されているステートメントを実行する直前に、プログラムを停止します。停止すると、プログラムウィンドウにブレイクポイントを含むルーチンを表示します。ブレイクポイントにカレントステートメント矢印が表示され、そのステートメントから実行できることを示します。

ここでは、以下の内容について説明します。

[ブレークポイントを設定](#)

[ブレークポイントを消去](#)

[一時的なブレークポイント](#)

[ブレークポイントを表示](#)

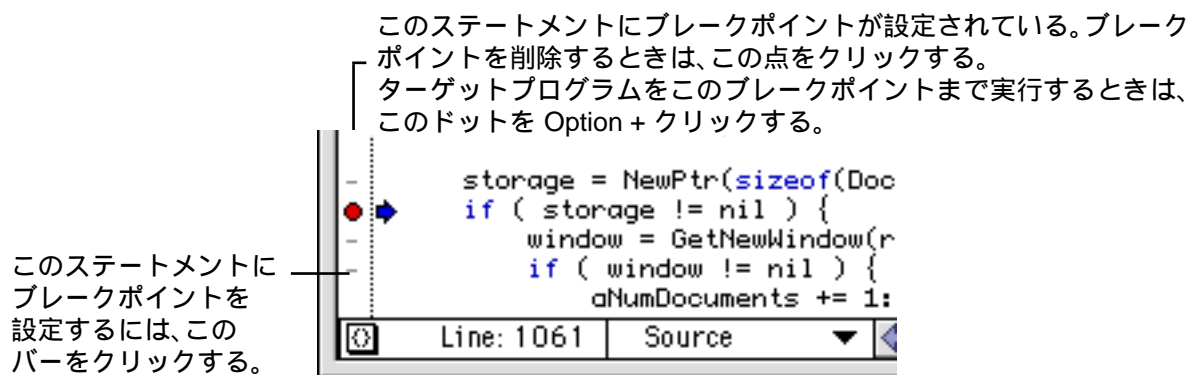
[条件ブレークポイント](#)

[ブレークポイントでコードを最適化](#)

## ブレークポイントを設定

プログラムウィンドウおよびブラウザウィンドウのソース欄で、マーカを使ってブレークポイントを設定できます。マーカは、ステートメントの左のブレークポイント欄の短いバーです(図 4.22)。バーは丸(カラーモニタでは赤)になります。これは、このステートメントにブレークポイントが設定されたことを示します。実行は、このステートメントが実行される直前で停止します。

図 4.22 ブレークポイントを設定



ヒント： コードの各行に一つのステートメントを書くようにしてください。ソースコードが読みやすいだけでなく、デバッグが簡単になります。ソースコードの一行に含まれているステートメントの数にかかわらず、ブレークポイントは一つしか設定できません。

## ブレークポイントを消去

一つのブレークポイントを消去するときは、ソース欄のブレークポイントの丸いマークをクリックします。マークはバーマークに変わり、ブレークポイントは消去されます。すべてのブレークポイントを消去するときは、デバッガの Debug メニューの『Clear All Breakpoints』コマンドを選択します。



## 一時的なブレークポイント

プログラムをあるステートメントまで実行して停止し、デバッグする作業を繰り返し行わないこともあります。一時的なブレークポイントを設定するには、停止したいステートメントの左のバーを Alt/Option + クリックします。実行を再開すると、そのステートメントまで実行されて停止します。

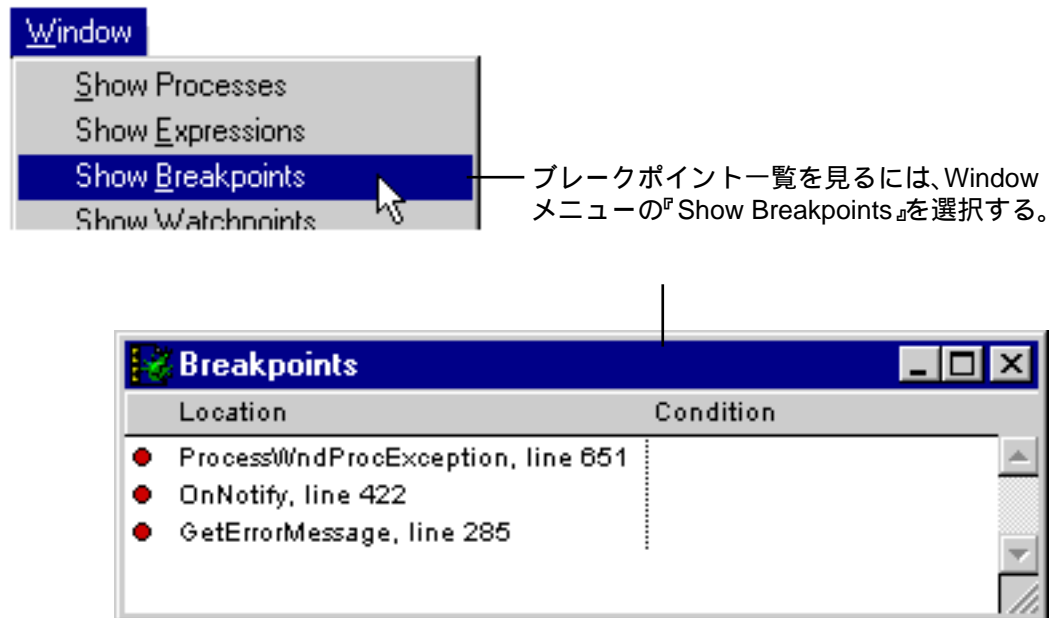
注意： Alt/Option + クリックしたところに既に普通のブレークポイントが設定されていると、そのブレークポイントは除去されますが、一時的なブレークポイントは有効です。

一時的なブレークポイントに着く前に、他のブレークポイントがある場合、プログラムは最初のブレークポイントで一時停止します。一時的なブレークポイントはまだ有効です。一時的なブレークポイントに実行が達した時に停止し、消去されます。

## ブレークポイントを表示

現在設定されているブレークポイントの一覧を見るには、デバッガの Window メニューの『Breakpoints Window』を選択してください。各ブレークポイントのソースファイルと行番号をリストしたウィンドウが表示されます(図 4.23)。ブレークポイントウィンドウ上でブレークポイントマーカをクリックすることで、ターゲットプログラムのブレークポイントをオン/オフに設定できます。

図 4.23 MW Debug で Breakpoints ウィンドウを表示



注意： ブレークポイントをダブルクリックすると、ブラウザウィンドウの上にそのブレークポイントに該当するソース部分を表示します。

[『Breakpoint ウィンドウ』\(p35\)](#) もご覧ください。

## 条件ブレークポイント

特定の位置で、指定した条件に合致したときに停止させる『条件ブレークポイント』を指定できます。条件ブレークポイントは、普通のブレークポイントに停止する条件式を付けたものです。コントロールがブレークポイントに達したとき、評価式の結果が TRUE (0 以外の値) であれば、そのブレークポイントでプログラムが停止します。このとき、評価式の結果が FALSE (結果の値が 0) であれば、プログラムはブレークポイントを無視して実行を続けます。

条件ブレークポイントは、ブレークポイントウィンドウで作成します。条件ブレークポイントを作成するには、以下の手順に従ってください。

1. ソース欄でブレークポイントを設定してください。
2. Window メニューの『Breakpoints Window』を選択して、ブレークポイントウィンドウを表示します。
3. ブレークポイントウィンドウで、ブレークポイントの Condition フィールドをダブルクリックして、評価式を入力します。または、ソースコードの表示または Expression ウィンドウから評価式をドラッグします。

[図 4.24](#) では、デバッガは行 120 の `NewBall()` ルーチンで、変数 `newTop` が 6 より大きいときだけ、実行を停止します。

図 4.24 条件ブレークポイントを設定



注意： 条件ブレークポイントは、ループ中で数回実行した後に停止したいときに有効な手段です。条件ブレークポイントをループの中に置いて、ループのインデックスが望みの値に達したら停止するように設定できます。

## ブレークポイントでコードを最適化

正確にブレークポイントを設定するために、デバッガはソースコードとオブジェクトコードの間の直接の対応関係に依存しています。コードの最適化はこの関係を壊し、ブレークポイントに問題を発生させるかもしれません。

ソースコードの左側にブレークポイントのバー ( - ) がない場合、その行にブレークポイントを設定することはできません。その理由を以下に示します。

その行のシンボル情報が利用できなくなる。

その行のルーチンは未使用であるため、リンカによって削除される。

コードは既に最適化されていて、最終的なオブジェクトコードと元のソースコードが対応していない。

例えば PowerPC コンパイラでは、『基本ブロック』の開始点に対応するソースの開始点にブレークポイントを設定できますが、そこには最低でも一つのインストラクションが含まれていなくてはなりません。

通常、ソースファイルのデバッグマーカがオンの場合、コンパイラは実際にコードを生成するソースの基本ブロックから開始させようとします。

---

```
- int i = 1;
- if (i)
-     {
-         int k;
-         int j = 1;
-         i = j;
-     }
```

---

{ のような行にはブレークポイントを設定できません。これはインストラクションを生成しないので、このソース行のユニークなオブジェクトコードアドレスがないためです。

一度でも最適化の設定をオンにすると、すべてが破壊されます。例えば『Instruction Scheduling』がオンの場合、コンパイラは各ソースステートメントの新しい基本ブロックを開始することはできなくなります。これはスケジューラーにブロック内の命令の並べ替えに最大限の柔軟性を与えます。ソースに対応する異なるインストラクションは不連続になるため、他のステートメントからのインストラクションと混同されてしまいます。上記の例では、ブレークポイントは次のようになります。

---

```
- int i = 1;
-     if (i)
-     {
-         int i;
-         int j = 1;
-         i = j;
-     }
```

---

最適化レベル 3 または 4 では、生成されたコードの中にソースステートメントが現れません。次のようなループになります。

```
i = 0;
j = 0;
while (i < 10)
{
    j = j + 1;
    i = i + 1;
}
```

コンパイラはこれをソースと等価のものに翻訳します。

```
j = j + 1; // duplicate 10 times
j = j + 1;
...
j = j + 1;
```

あるいは次のようにします。

```
j = 10;
```

そしてループに対応するインストラクションをすべて削除します。

シンボルデバッグで最良の結果を得るためには、最適化をオフにするか、デバッグに危険のない程度の最適化だけを行ってください。ターゲットによって可能な最適化は異なります。詳細は各『Targeting』マニュアルを参照してください。通常は『Peephole』、『Global Optimizers』、『Instruction Scheduling』をオフにし、『Don't Inline』をオンにします。これで重要なステートメントにブレークポイントを設定できます。ほとんどのターゲットにこの最適化を適用できます。Edit メニューの『Target Settings』を選択すると、プロジェクトに対する最適化を見ることができます。このコマンドにはビルドターゲットの名前が入ります。

ブレークポイントを設定した後で、ブラウザウィンドウまたはプログラムウィンドウからプログラムを実行できます。Control メニューの『Step』、『Step Over』、『Step Into』、『Run』を使います。これらのコマンドは MW Debug でも使用できます。

[『コードの実行、ステップ実行、停止』\(p49\)](#) を参照してください。

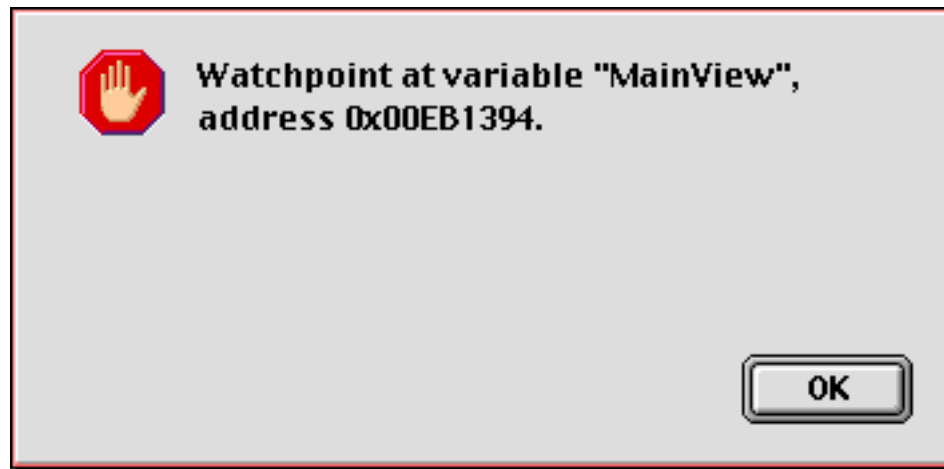
## ウォッチポイント

ウォッチポイントは、メモリ上で注視したい位置または範囲です。この領域に新しい値が書き込まれるとデバッガは停止し、アラートを表示します (図 4.25)。その後はデバッガにコントロールが戻るので、普通にデバッガのコマンドを使ってコールチェーンをたどったり、変数の値を表示、変更したり、ステップ実行したりできます (特に、デバッガレベル

では、ウォッチポイントをトリガする位置の内容を、再度ウォッチポイントをトリガすることなく変更できます。『Run』コマンド（またはツールバーの [Run] ボタン）を使って実行を再開できます。

Mac OS :ウォッチポイント機能を使うためには、System 7.5 以上の MacOS が必要です。また、68K Macintosh では、仮想メモリをオンに設定しないとウォッチポイント機能が動作しません。ウォッチポイント機能は、Speed Doubler や RAM Doubler とは互換性がありません。

図 4.25 ウォッチポイントの警告



## ウォッチポイントを設定

以下のいずれかの操作で、ウォッチポイントを設定できます。

変数ウィンドウ、またはブラウザウィンドウの大域変数欄で変数を選択し、Debug メニューの『Set Watchpoint』を選びます。

他のウィンドウから変数を Watchpoints ウィンドウヘドラッグします。

メモリウィンドウのある範囲を選択し、Debug メニューの『Set Watchpoint』を選びます。

ウォッチポイントが設定された変数またはメモリ範囲は、シンボルウィンドウ、変数ウィンドウ、メモリウィンドウ上で下線付きの赤字で示されます。

[『Use Syntax Coloring in Source Display』\( p97 \)](#) を参照してください。

**警告！** ウォッチポイントを指定できるメモリ領域に、若干の制限があります。大域変数、およびアプリケーションヒープにとられるオブジェクトのメモリ領域上だけに、ウォッチポイントを設定できます。スタックベースの局所変数やレジスタ変数にはウォッチポイントを設定できません。

Mac OS :ローメモリやシステムヒープにはウォッチポイントを設定できません。

注意： 小さな 68K プロジェクトのデバッグ中に、大域変数にウォッチポイントを設定しているとき、『スタック上にはウォッチポイントを設定できません。』というメッセージが表示されることがあります。これは 68K のランタイムアーキテクチャによる制限です。

## ウォッチポイントを消去

以下のいずれかの操作でウォッチポイントを消去できます。

ウォッチポイントで停止した後、Debug メニューの『Clear Current Watchpoint』を選びます。

変数ウィンドウ、または MW Debug のブラウザウィンドウの大域変数欄で変数を選択し、Debug メニューの『Clear Watchpoint』を選びます。

メモリウィンドウから範囲を選択し、Debug メニューの『Clear Watchpoint』を選びます。

Watchpoints ウィンドウで既存のウォッチポイントを選択した後、以下のいずれかの操作を行います。

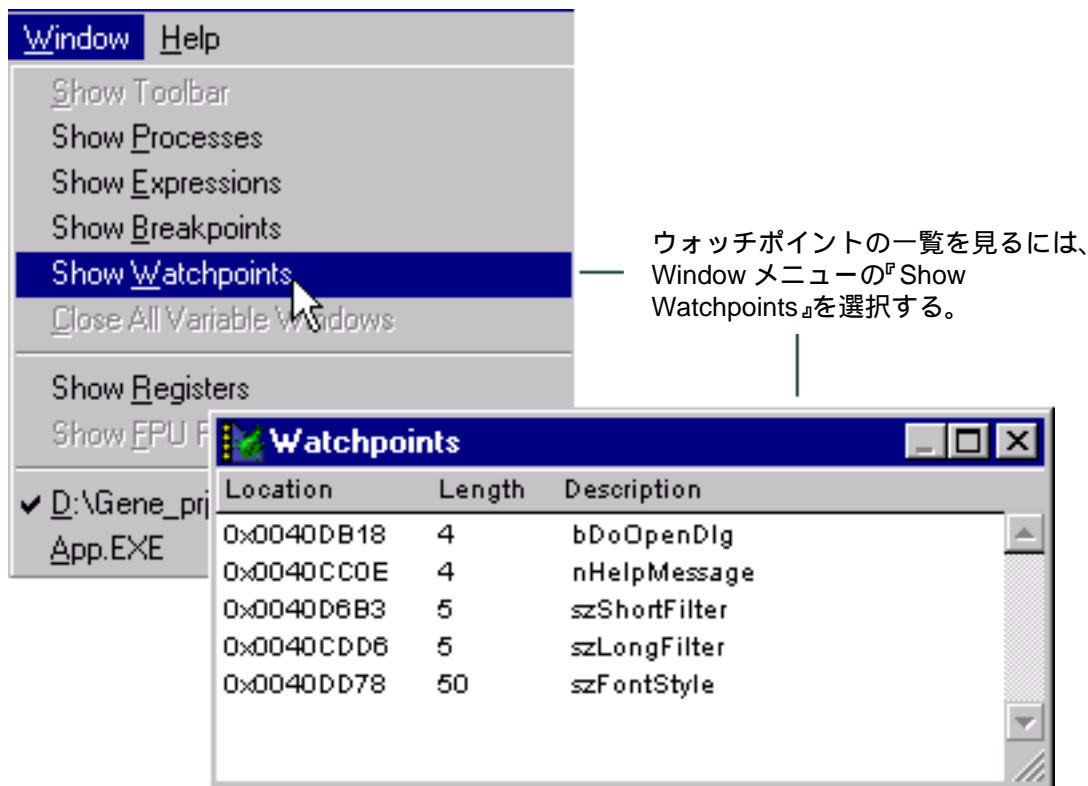
- Debug メニューの『Clear Watchpoint』を選ぶ。
- Edit メニューの『Clear』を選ぶ。
- Backspace/Delete キーを押す。

ターゲットプログラムが終了または中止されたとき、すべてのウォッチポイントは自動的に消去されます。

## ウォッチポイントを表示

現在設定されているウォッチポイントの一覧を表示するには、Window メニューの『Watchpoints Window』を選びます。各ウォッチポイントのアドレスと長さの一覧ウィンドウが表示されます ( [図 4.26](#) )。

図 4.26 MW Debug の Watchpoints ウィンドウを表示



[『Watchpoint ウィンドウ』\(p36\)](#) を参照してください。

## データの表示、変更

デバッガの重要な機能に、変数の現在値を表示したり、必要に応じてその値を変更できる機能があります。ここでは、変数を見たり、変更する方法について説明します。以下の内容について説明します。

[局所変数を表示する](#)

[大域変数を表示する](#)

[新しいウィンドウへデータを入れる](#)

[データ型を表示する](#)

[データを異なるフォーマットで表示する](#)

[データを異なる型で表示する](#)

[変数の値を変更する](#)

[Expression ウィンドウを使う](#)

[メモリダンプを表示する](#)

### [アドレスを指定してメモリを表示する](#)

### [プロセッサレジスタを表示する](#)

特定のターゲットのデータを表示または変更するための詳細は、それぞれの『Targeting』マニュアルをご覧ください。

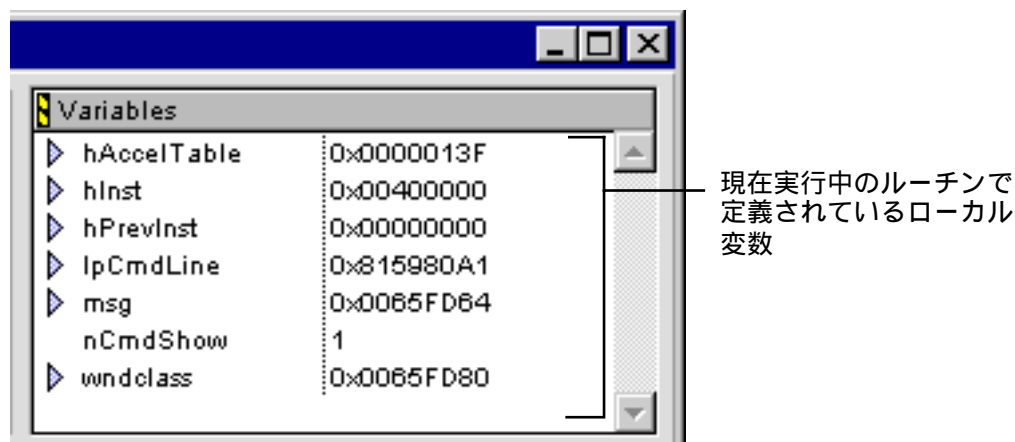
## 局所変数を表示する

局所変数は、プログラムウィンドウの変数欄に表示されます（[図 4.27](#)）。変数がハンドル、ポインタ、または構造体の場合、名前の左にある拡張ボタンをクリックすると、表示を拡張してさらに詳しい情報（構造体のメンバ、ポインタやハンドルによって参照されているデータ）を見ることができます。

MW Debug の Data メニューの『Expand』または『Collapse All』コマンドでも、変数の表示を拡張または縮小できます。

[『Expand』（p113）](#)、[『Collapse All』（p113）](#) および [『変数欄』（p23）](#) もご覧ください。

図 4.27 局所変数を表示する



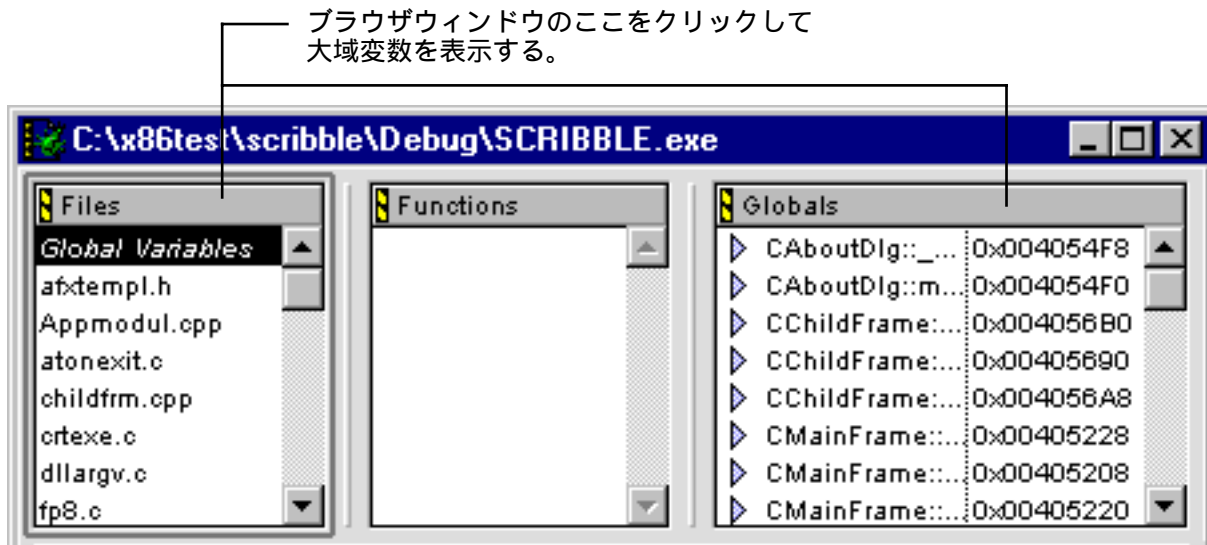
## 大域変数を表示する

大域変数は、プログラムウィンドウおよび MW Debug のブラウザウィンドウに表示されます（[図 4.28](#)）。大域変数は、プログラムウィンドウでは変数欄の点線の下に表示され、MW Debug のブラウザウィンドウではファイル欄で『Global Variables』を選択すると、大域変数欄に表示されます。

[『変数欄』（p23）](#)、[『大域変数欄』（p33）](#) および [『大域変数欄』（p33）](#) もご覧ください。



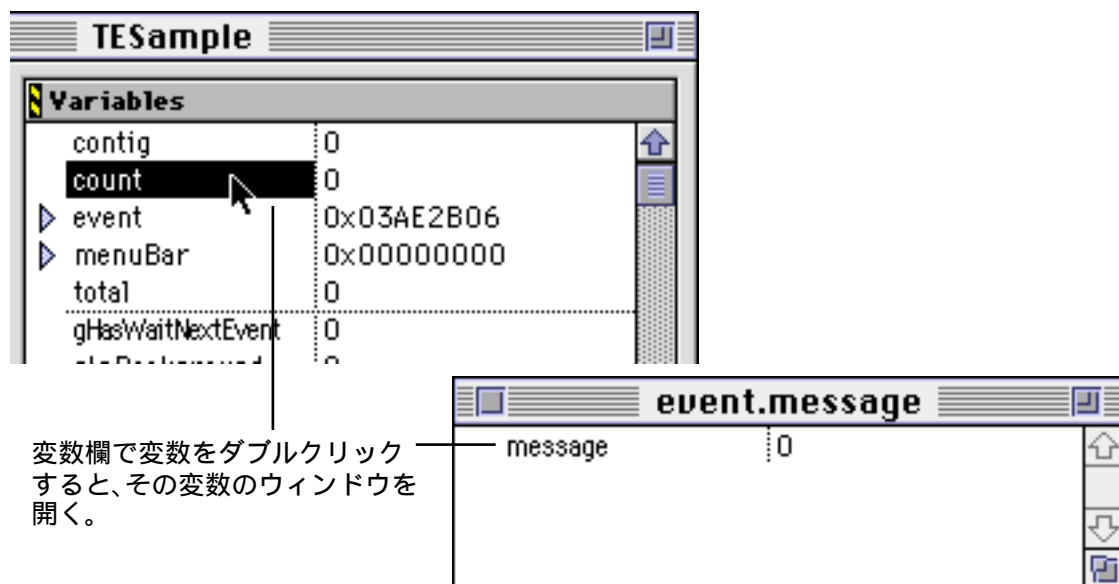
図 4.28 MW Debug のブラウザウィンドウに大域変数を表示する



### 新しいウィンドウへデータを入れる

局所変数欄や大域変数欄が、データを表示するのにいつも最適な場所とは限りません。変数または変数のグループを別のウィンドウまたは専用のウィンドウに表示することができます。

図 4.29 変数を専用のウィンドウに表示する



変数またはメモリ位置を専用のウィンドウに表示するには、変数をダブルクリックします（図 4.29）。または、名前を選択して Data メニューの『View Variable』を選択します。変数

が配列の場合は、代わりに『View Array』を使います。変数が置かれているメモリをメモリダンプとして表示するには、『View Memory』または『View Memory as』コマンドを使います。

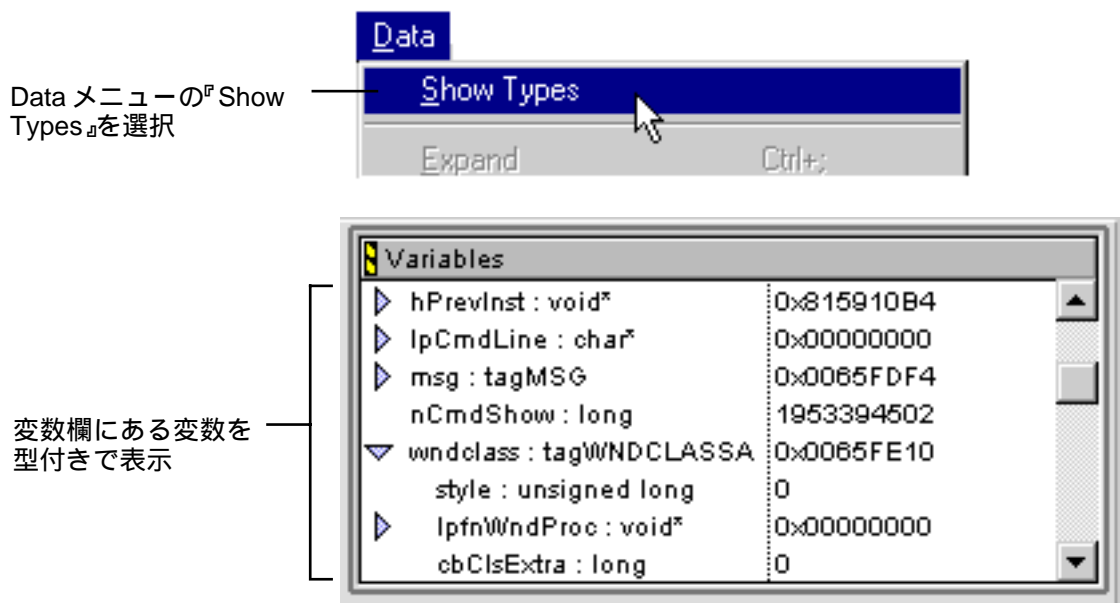
[『変数ウィンドウ』\(p37\)](#)、[『配列ウィンドウ』\(p38\)](#) および [『メモリウィンドウ』\(p39\)](#) もご覧ください。

## データ型を表示する

デバッガで、変数のデータ型をウィンドウごとに表示設定できます。データ型を表示したいウィンドウまたは欄を選択して、Data メニューの『Show Types』を選択します。ウィンドウまたは欄内のデータ型の後に、変数名とメモリ位置が表示されます ( [図 4.30](#) )。

ヒント： デバッガが起動したときに自動的にデータ型を表示する設定にするには、Debugger Display Settings 環境設定パネルで『In variable panes, show variable types by default』をオンにします。詳しくは、[『Settings 環境設定パネル』\(p93\)](#) をご覧ください。

図 4.30 データ型を表示する



## データを異なるフォーマットで表示する

変数の値を表示するフォーマットをコントロールできます。以下のフォーマットで表示できます。

符号付き 10 進数

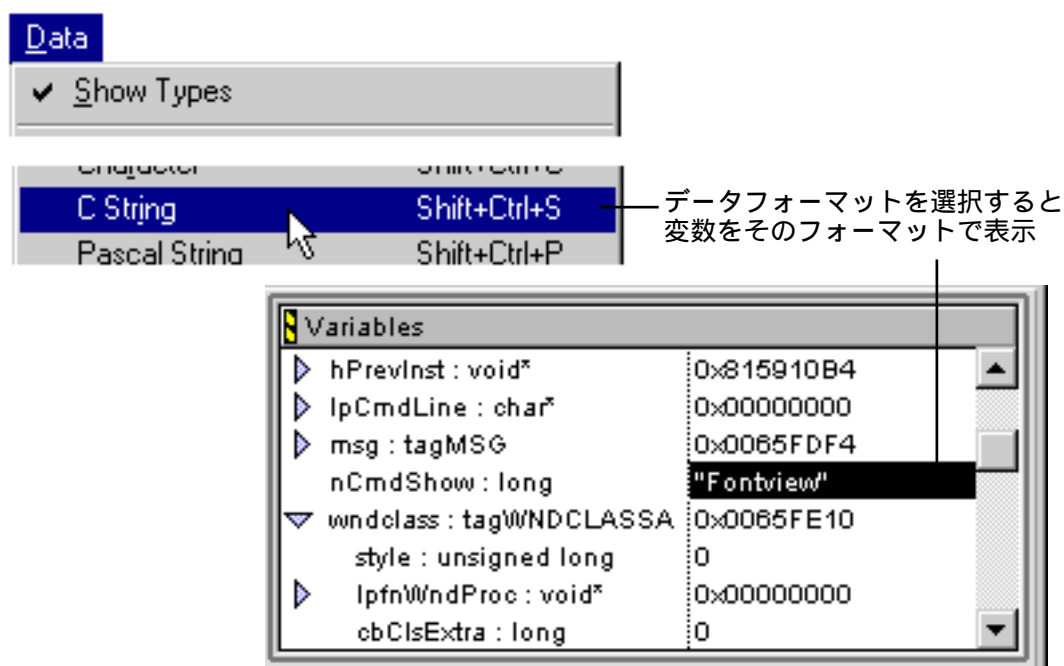
符号なし 10 進数

16 進数

文字  
C 文字列  
Pascal 文字列  
浮動小数点数  
enum 型 ( 列挙型 )  
固定小数点数  
fract 型

特定のフォーマットでデータを表示するには、表示したいウィンドウで変数名または変数の値を選択し、Data メニューから表示したいフォーマットを選択します ( 図 4.31 )。

図 4.31 データフォーマットを選択する



すべてのデータ型に対してすべてのフォーマットが使えるわけではありません。例えば、変数が整数値 ( short 型、 long 型 ) のときは、符号付き 10 進数、符号なし 10 進数、16 進数、文字、または文字列として表示できますが、浮動小数点数、固定小数点数、fract 型のフォーマットでは表示できません。

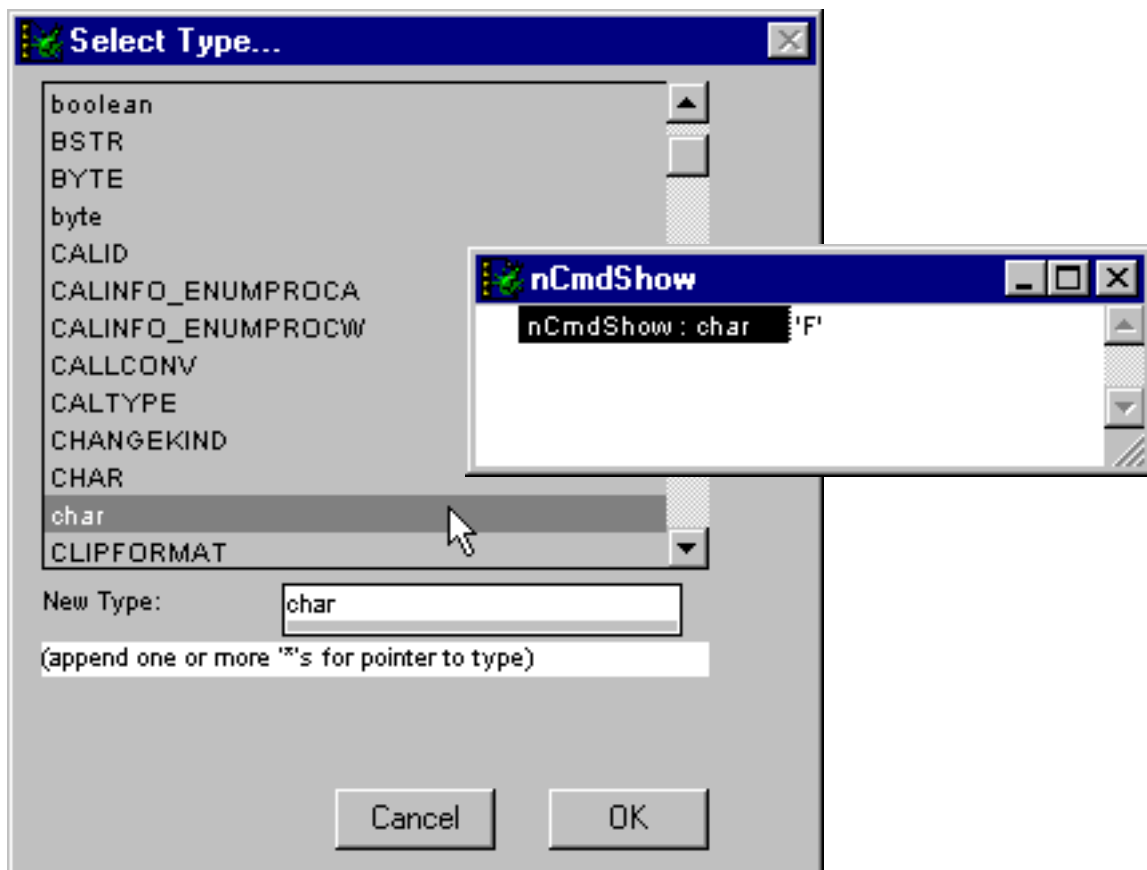
## データを異なる型で表示する

変数、レジスタ、またはメモリを表示するデータ型を変更するには、Data メニューの『View as』を使います。以下の手順に従ってください。

1. データ型を変更したい変数を、ウィンドウまたは変数欄で選択します。

2. Data メニューで『View as』コマンドを選択します。  
ダイアログ ( 図 4.32 ) が表示されます。
3. 表示したいデータ型を選択します。  
選択したデータ型の名前がダイアログの下の方の『New Type』ボックスに表示されます。  
ポインタとして表示したい場合は、星印 ( \* ) を型名に付けます。
4. [ OK ] ボタンをクリックします。  
表示されている値の型が、指定された型に変わります。

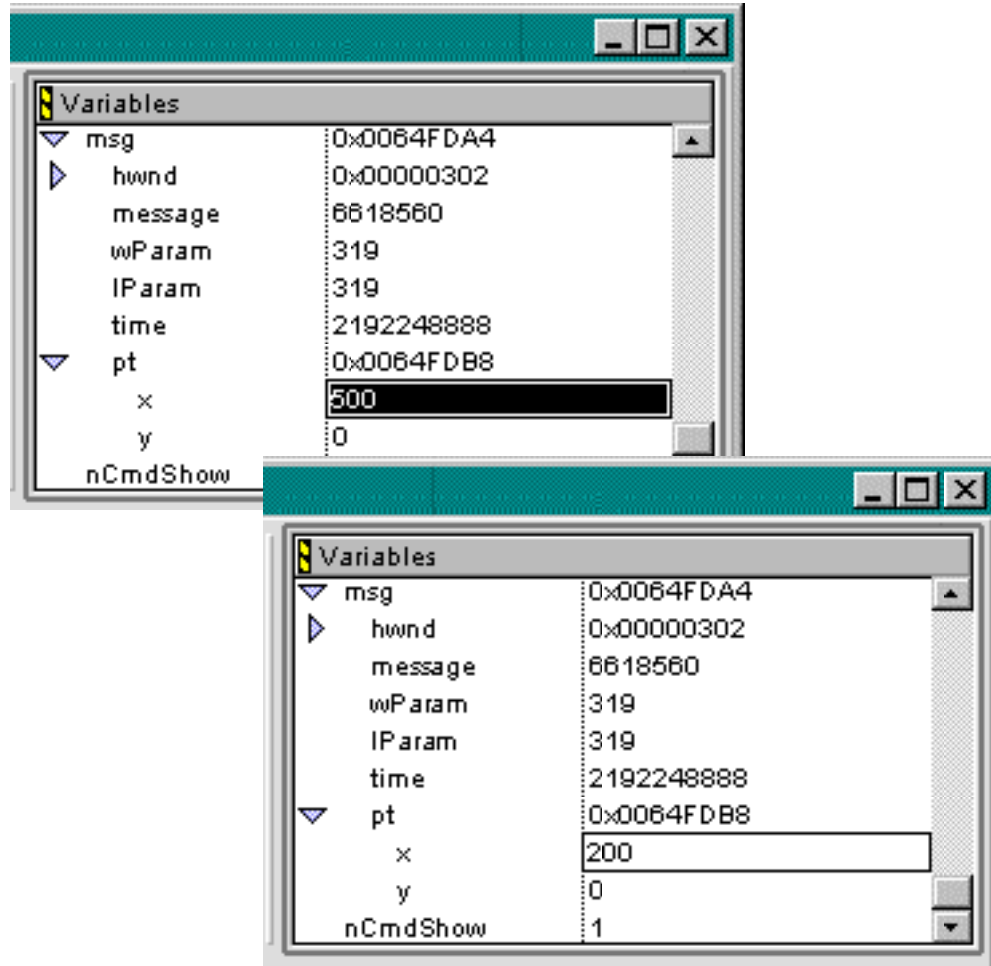
図 4.32 データ型を選択する



### 変数の値を変更する

変数が表示されているウィンドウであればどこでも ( プログラムウィンドウの局所変数欄、ブラウザウィンドウの大域変数欄、または変数ウィンドウ、配列ウィンドウ、Expression ウィンドウ ) その値を変更できます。古い値をダブルクリックして新しい値を入力します ( 図 4.33 )。

図 4.33 変数の値を変更



変数の値は、下記のいずれかのフォーマットで入力できます。

- 10 進数
- 16 進数
- 浮動小数点数
- C 文字列
- Pascal 文字列
- 文字定数

文字列および文字定数を入力するには、C 形式のクォートが必要です（文字定数にはシングルクォート ' '、文字列にはダブルクォート " "）。Pascal 文字列は、\p を文字列の先頭に含まなければなりません。

**警告!** 変数の値を変更することは、トラブルの原因となりますので注意してください。デバッガでは、変数を適切なデータ型のどのような値にも設定できます。例えば、ポインタを `nil` に設定することもできますが、システムがクラッシュします。

## Expression ウィンドウを使う

Expression ウィンドウは、よくアクセスする局所変数や大域変数、構造体、配列をまとめて一つのウィンドウで見ることができます。Expression ウィンドウを表示するには、デバッガの Window メニューの『Expressions Window』コマンドを選択します。Expression ウィンドウに変数を追加するには、他のウィンドウからドラッグ & ドロップするか、または別の変数を選択して Data メニューの『Copy to Expression』を選択します。

デバッガで実行を停止すると、Expression ウィンドウの内容が更新されます。実行の範囲外の変数は空白表示されます。Expression ウィンドウを利用して以下のようなことができます。

ルーチンの局所変数を、その内容を表示するために拡張する前に Expression ウィンドウへ入れておきます。そのルーチンが終了すると変数は Expression ウィンドウに残り、ルーチンに実行が戻ると拡張されたままになります。Expression ウィンドウは、ルーチンが終了してその変数が使用範囲の外に出てしまっても、拡張した変数表示が自動的に縮小されることはありません（局所変数欄などは変数が使用範囲の外に出ると変数表示は自動的に縮小されます）。

同じデータ項目のコピーを複数作成し、Data メニューの『Copy to Expression』と『View as』コマンドで、異なるデータ型を設定して表示しておくことができます。

リストを並べ変えることができます。Expression ウィンドウ内で項目をドラッグすることで順序を変えることができます。

局所変数を呼出しルーチンから表示できます。呼び出し元の局所変数を表示するためにコールチェーンをさかのぼる必要はありません（これを行うと、現在実行中のルーチンの局所変数が隠されてしまいます）。呼び出し元の局所変数を Expression ウィンドウに追加することにより、コールチェーン欄のレベルを変更せずに表示することができます。

[『Expression ウィンドウ』\(p35\)](#) もご覧ください。

## メモリダンプを表示する

ローメモリの値を見たり、変更したりするときは以下の手順にしたがってください。

1. 表示したいメモリのアドレスを示す項目または評価式を選択します。
2. Data メニューの『View Memory』を選択します。

新しいメモリウィンドウが表示され、16 進数および ASCII 文字でメモリの内容を表示します。このメモリウィンドウ上で、16 進数または ASCII 文字を入力してメモリの内容を直接変更できます。さらに、ウィンドウの一番上のテキストフィールドで、表示したいメモリの先頭アドレスを変更することもできます。

## アドレスを指定してメモリを表示する

Data メニューの『View Memory』および『View Memory as』コマンドで、任意のアドレス（レジスタに格納されたアドレスも可）で指されたメモリの現在値を表示できます。変数またはレジスタが指しているメモリを表示するには、以下の手順に従ってください。

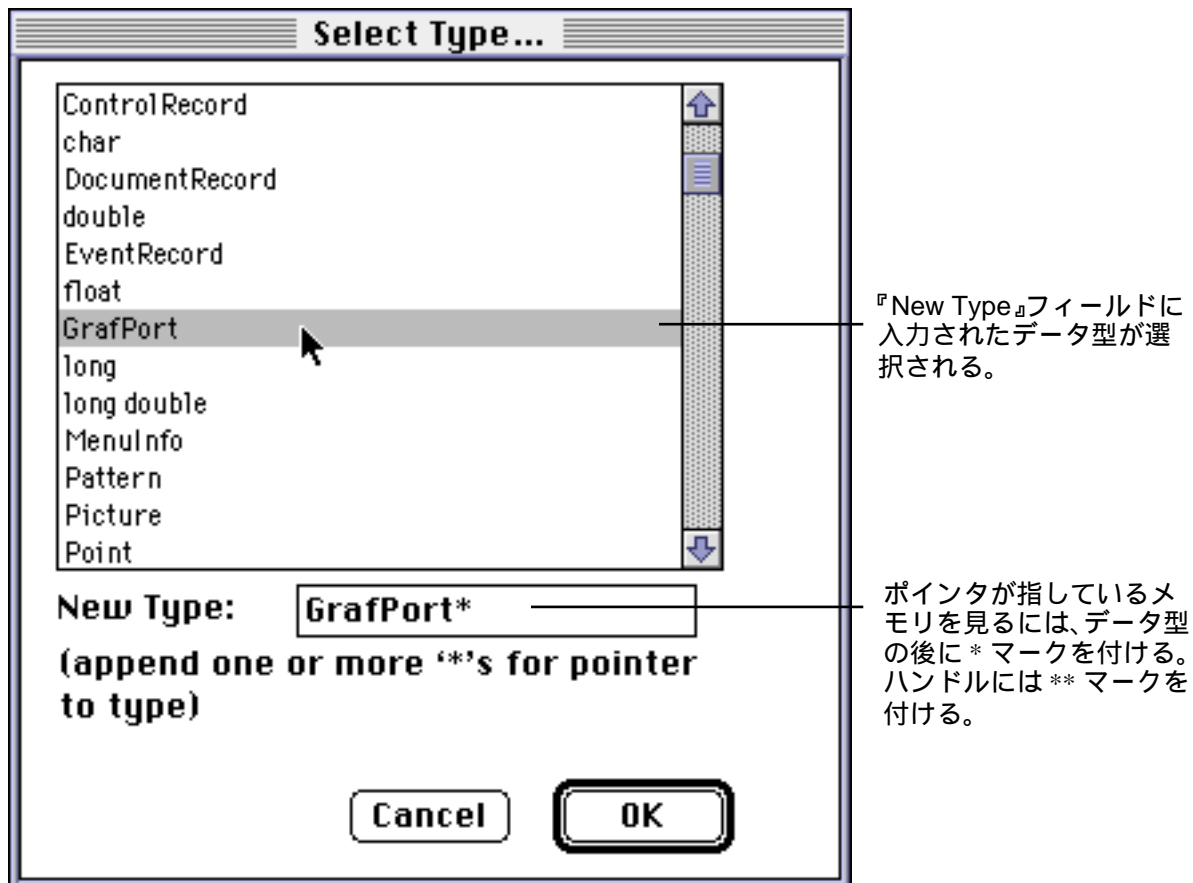
1. 表示したいウィンドウで変数値またはレジスタを選択します。
2. Data メニューで『View Memory』または『View Memory as』コマンドを選択します。

『View Memory』を選択すると、ポインタによって参照されたアドレスから始まるメモリの内容を表示するウィンドウを開きます。『View Memory as』を選択すると、データ型を選択するダイアログが表示されるので（[図 4.34](#)）ステップ 3 へ進んでください。

3. 『View Memory as』を選択した場合は、ダイアログでデータ型を選択します。

選択したデータ型の名前がダイアログの下の方の『New Type』ボックスに表示されます。レジスタが指しているメモリを表示するには、型名に星印（\*）を付けます。

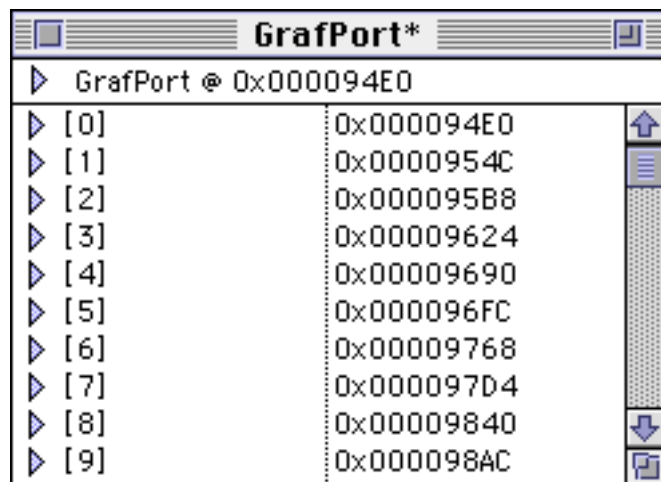
図 4.34 表示するメモリのデータ型を選択する



4. [OK] ボタンをクリックします。

ポインタで指されたアドレスを先頭アドレスとするメモリの内容を表示する、新しいウィンドウが表示されます ( [図 4.35](#) )。

図 4.35 指定したデータ型でメモリを表示する



注意： 同じ方法を使って、スタックの内容を表示することができます。ターゲットプロセッサがスタックポインタをレジスタに格納しているのならば、そのレジスタの値を選択します。その後は上記の手順と同じです。

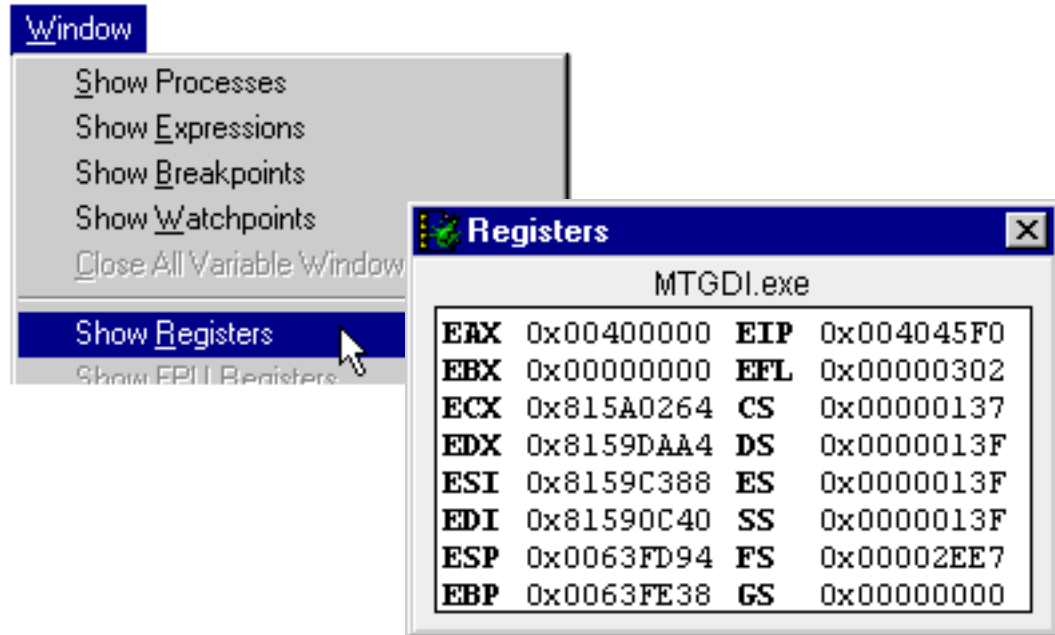
[『メモリウィンドウ』\(p39\)](#) もご覧ください。

## プロセッサレジスタを表示する

プロセッサレジスタの内容を表示するには、MW Debug の Window メニューで『Show Registers』または『Show FPU Registers』コマンドを選択します ( [図 4.36](#) ) ( FPU を持たないターゲットでは FPU レジスタウィンドウは利用できません )。



図 4.36 MW Debug でプロセッサレジスタを表示する



[『レジスタウィンドウ』\(p41\)](#) を参照してください。

## ソースコードを編集

デバッガから直接コードを編集することはできません。しかし、デバッガでファイルを開いてコードを変更することは可能です。ブラウザウィンドウのファイル欄で以下の操作をしてください。

ファイル名をダブルクリックする。

ファイル名を選択し、デバッガの File メニューの『Edit filename』を選択する。

これによってエディタウィンドウにファイルを開くので、コードの編集ができます。

Windows :これで、指定したエディタにファイルが開かれます。[『Win32 Settings 環境設定パネル』\(p102\)](#) を参照してください。





## 第 5 章 評価式

CodeWarrior デバッガの Expression ( 評価式 ) は、数式や論理式の値を表示したり、ブレークポイントの条件式を作成するために使われます。

### 評価式の概要

評価式は値を生成する計算式を表します。デバッガは、Expression ウィンドウに値を表示したり、ブレークポイントウィンドウ内のブレークポイントに付けられた値に基づいて動作します。一つのステートメントを実行するたびに、すべての評価式が評価されます。

評価式は、文字変数 ( 数字と文字列 )、変数、レジスタ、ポインタ、および C++ オブジェクトメンバ、さらに加減算、論理 AND、等号などのオペレータにより構成されます。

評価式は、Expression ウィンドウ、ブレークポイント、またはメモリウィンドウで使われます。デバッガは評価式の結果を各ウィンドウで別の用途に使います。

この章ではデバッガがどのように評価式を扱うのかを説明します。

[評価式の翻訳](#)

[評価式を使う](#)

[評価式の例](#)

[評価式のシンタックス](#)

### 評価式の翻訳

ここではデバッガがどのように各ウィンドウの評価式を翻訳するのかを説明します。

[Expression ウィンドウの評価式](#)

[ブレークポイントウィンドウの評価式](#)

[メモリウィンドウの評価式](#)

#### Expression ウィンドウの評価式

Expression ウィンドウは、評価式とその結果の値を表示します。評価式の値を見るには、Expression ウィンドウに評価式を入力します。新しい評価式を入力するには、以下の手順にしたがってください。

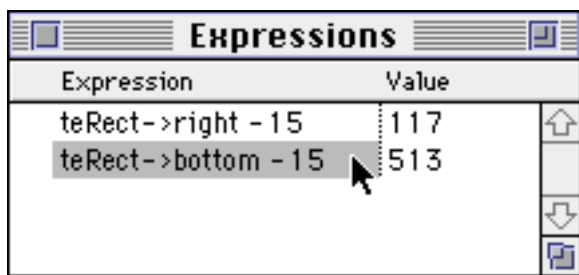
1. Expression ウィンドウを表示します。

Window メニューの『Expression Window』を選択します。または開いている Expression ウィンドウをクリックしてアクティブにします。

2. Data メニューで『New Expression』を選択します。
3. 新しい評価式を入力し、Enter キーまたは Return キーを押します。

評価式 (Expression 列) の値が評価式の横 (Value 列) に表示されます (図 5.1)。他のウィンドウにある評価式を Expression ウィンドウへドラッグして、新しい評価式を作成することもできます。

図 5.1 Expression ウィンドウの評価式の例



Expression	Value
teRect->right - 15	117
teRect->bottom - 15	513

Expression ウィンドウでは、評価式は数式として扱われます。デバッガは評価式の結果を論理値として扱いません (ただしブレークポイントウィンドウでは論理式として扱います)。

[『Expression ウィンドウ』\(p35\)](#) もご覧ください。

## ブレークポイントウィンドウの評価式

ブレークポイントウィンドウで、ブレークポイントの条件式を入力することができます。ブレークポイントウィンドウでは、評価式は論理式として扱われます。評価式の結果がゼロの場合は FALSE と解釈され、デバッガはそのブレークポイントを無視して実行を続けます。評価式の結果がゼロ以外の値の場合は TRUE と解釈され、ブレークポイント ( ) が設定されていたら、そのブレークポイントで実行を一時停止します。

ブレークポイントの設定方法については、[『ブレークポイントを設定』\(p64\)](#) をご覧ください。ブレークポイントを設定した後、ブレークポイントに評価式を付けて、その評価式を条件にすることができます。

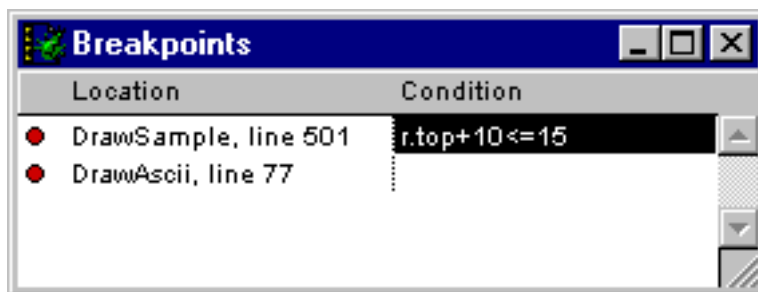
1. ブレークポイントウィンドウを表示します。

Window メニューの『Expression Window』を選択するか、開いているブレークポイントウィンドウをクリックしてアクティブにします。

2. 条件を設定します。

Condition 欄をダブルクリックした後、そこへ評価式を入力します (図 5.2)。他のウィンドウにある評価式をブレークポイントウィンドウの Condition 欄へドラッグ & ドロップして、評価式を追加または変更することもできます。

図 5.2 ブレークポイントウィンドウの評価式



条件ブレークポイントは、実行がブレークポイントに達したときに評価式の結果が TRUE（ゼロ以外）であれば、プログラムを停止します。評価式の結果が FALSE（ゼロ）であれば実行は停止せずに続きます。

[『Breakpoint ウィンドウ』\(p35\)](#) および [『条件ブレークポイント』\(p66\)](#) もご覧ください。

## メモリウィンドウの評価式

メモリウィンドウでは、評価式はアドレスとして扱われます。ウィンドウの一番上のテキストフィールド内の評価式が、ウィンドウに表示するメモリ先頭アドレスを定義します。メモリウィンドウの先頭アドレスを変更するには、以下の手順に従ってください。

1. メモリウィンドウを表示します。

Data メニューの『View Memory』を選択します。または、開いているメモリウィンドウをクリックしてアクティブにします。

2. 新しい評価式を入力します。

評価式フィールドをダブルクリックし、新しい評価式を入力します。または、他のウィンドウにある評価式をメモリウィンドウの先頭アドレスフィールドへドラッグして、新しい評価式を作成することもできます。

新しい評価式の結果が示すアドレスを先頭アドレスとして、メモリの内容がダンプされます。

[『メモリウィンドウ』\(p39\)](#) もご覧ください。

## 評価式を使う

CodeWarrior デバッガの評価式シンタックスは、C/C++ の評価式のシンタックスと似ています。C/C++ の評価式シンタックスを少し拡張し、若干の制限もあります。Pascal 形式の評価式もサポートしています。

### 特別な評価式の機能

評価式は特別な項目を参照できます。

デバッガは int 型を 4 バイト長とみなします。2 バイト整数値としたい場合は short 型を使います。

デバッガは double 型を 8 バイト (64 ビット) ではなく 10 バイト (80 ビット) とみなします。

文字列を比較するには、== (等号) または != (不等号) オペレータを使います。デバッガは、Pascal 文字列と C 文字列を区別しています。Pascal 文字列を比較するときは、文字列の先頭に "\p" を付けてください。次の評価式は、C 文字列と Pascal 文字列を比較しているので、結果は FALSE になります。

```
"Nov shmoz ka pop" == "\pNov shmoz ka pop"
```

(Mac OS) レジスタの値を参照するには、® シンボルとレジスタ名を使います (® シンボルを入力するには、Option + R キーを入力します)。

## 評価式の制限

デバッガの評価式に適用される制限事項を以下に示します。

C/C++ プリプロセッサの定義およびマクロ (#define 命令で定義) は使用できません。たとえそれらがソースコードの中で定義されていたとしても、評価式の中では使用できません。

副作用があるオペレータは使えません。増分オペレータ (i++)、減分オペレータ (i--)、代入 (i = j) は使えません。

関数呼び出しは使用できません。

関数名や関数へのポインタは使用できません。

評価式のリストは使用できません。

C++ クラスメンバに対してのポインタは使用できません。

デバッガは、ネストされたブロックで使われている同じ名前の変数を区別できません ([リスト 5.1](#) をご覧ください)。

### リスト 5.1 同じ名前の変数がネストしたブロックにある例 (C++)

---

```
// The debugger can't distinguish between x the
// int variable and x the double variable. If x
// is used in an expression, the debugger won't
// know which one to use.

void f(void)
{
    int x = 0;
    ...
    {
        double x = 1.0;
        ...
    }
}
```

```
    }  
}
```

デバッガで使えない型宣言は、使えません ([リスト 5.2](#))。

#### リスト 5.2 評価式の型宣言 (C/C++)

```
// Use long in expressions; Int32 not available  
typedef long Int32;  
  
// Use Rect* in expressions; RectPtr not  
// available  
typedef Rect* RectPtr;
```

ネストした型情報は使えません。[リスト 5.3](#) ではデバッガ評価式で、`Outer::Inner` ではなく `Inner` を使います。

#### リスト 5.3 ネストした型情報 (C/C++)

```
// To refer to the i member, use Inner.i,  
// not Outer::Inner.i  
  
struct Outer  
{  
    struct Inner  
    {  
        int i;  
    };  
};
```

## 評価式の例

以下に評価式が入力可能なウィンドウで入力できる評価式の簡単な例を示します。

10 進数の定数を直接入れます。

`160`

16 進値の値は先頭に『0x』を付けて入れます。

`0xA0`

変数の値を求めます。

`myVariable`

変数を 4 ビット左へシフトした値を求めます。

`myVariable << 4`

二つの変数の差を求めます。

`myRect.bottom - myRect.top`

二つの数値変数のうち大きい方の値を返します。

```
(foo > bar) ? foo : bar
```

ポインタ変数で指されている変数の値を求めます。

```
*MyTablePtr
```

構造体のサイズ（コンパイル時に決定）を求めます。

```
sizeof(myRect)
```

ポインタ変数->で指されている構造体のメンバの値を求めます。

```
myRectPtr->bottom
```

または

```
(*myRectPtr).bottom
```

オブジェクトのクラスメンバの値を求めます。

```
myDrawing::theRect
```

以下に論理式の例を示します。結果は、ゼロ以外を TRUE、ゼロを FALSE と解釈します。

変数が 0 であることを調べます。

```
!isDone
```

または

```
isDone == 0
```

変数がゼロでないことを調べます。

```
isReady
```

または

```
isReady != 0
```

数値変数が他の数値変数の値以上であることを調べます。

```
foo >= bar
```

数値変数が他の二つの数値変数より小さいことを調べます。

```
(foo < bar) && (foo < car)
```

変数の 4 ビット目が 1 であることを調べます。

```
((char)foo >> 3) & 0x01
```

C 文字列変数を C 文字列定数と比較します。

```
cstr == "Nov shmoz ka pop"
```

Pascal 文字列変数を Pascal 文字列定数と比較します。

```
pstr == "\pScram gravy ain't wavy"
```

常に TRUE です。

```
1
```

常に FALSE です。

```
0
```

## 評価式のシンタックス

この節では、デバッガの評価式（Expression）のシンタックスについて説明します。定義の最初の行は、これから定義する項目です。字下げされている項目は、定義の内容を示しま



す。複数の定義を持つ項目は、それぞれの定義が一行ずつ表示されます。山括弧 (<>) でくくられた項目は、他で定義されています。斜体の項目は、値またはシンボルによって置き換える項目です。その他の項目は、すべてこのまま書かなければなりません。

例えば、

```
<name>
    identifier
    <qualified-name>
```

この例では、『name』を定義します。『name』は『*identifier*』または『*qualified-name*』となることができます。後者は、ここでリストされている他の定義でシンタックスを定義されています。

```
<name>
    identifier
    <qualified-name>

<typedef-name>
    identifier

<class-name>
    identifier

<qualified-name>
    <qualified-class-name>::<name>

<qualified-class-name>
    <class-name>
    <class-name>::<qualified-class-name>

<complete-class-name>
    <qualified-class-name>
    :: <qualified-class-name>

<qualified-type-name>
    <typedef-name>
    <class-name>::<qualified-type-name>

<simple-type-name>
    <complete-class-name>
    <qualified-type-name>
    char
    short
    int
    long
    signed
    unsigned
    float
    double
    void

<ptr-operator>
    *
    &
```

```
<type-specifier>
    <simple-type-name>

<type-specifier-list>
    <type-specifier> <type-specifier-list>(opt)

<abstract-declarator>
    <ptr-operator> <abstract-declarator>(opt)
    (<abstract-declarator>)

<type-name>
    <type-specifier-list> <abstract-declarator>(opt)

<literal>
    integer-constant
    character-constant
    floating-constant
    string-literal

<register-name>
    ®PC
    ®SP
    ®Dnumber
    ®Anumber

<register-name>
    ®Rnumber
    ®FPRnumber
    ®RTOC

<register-name>
    $PC
    $SP
    $RTOC
    $Anumber
```

---

注意： プロセッサがターゲットにしていないレジスタは、未知のレジスタ評価式の値を表示できません。

---

---

注意： レジスタを指定する場合、値の範囲はターゲットプロセッサで利用可能なレジスタの数によって異なります。

---

```
<primary-expression>
    <literal>
    this
    ::identifier
    ::<qualified-name>
    (<expression>)
    <name>
    <register-name>
```

```
<postfix-expression>
    <primary-expression>
    <postfix-expression>[<expression>]
    <postfix-expression>.<name>
    <postfix-expression>-><name>

<unary-operator>
    *
    &
    +
    -
    !
    ~

<unary-expression>
    <postfix-expression>
    <unary-operator> <cast-expression>
    sizeof <unary-expression>
    sizeof(<type-name>)

<cast-expression>
    <unary-expression>
    (<type-name>)<cast-expression>

<multiplicative-expression>
    <cast-expression>
    <multiplicative-expression> * <cast-expression>
    <multiplicative-expression> / <cast-expression>
    <multiplicative-expression> % <cast-expression>

<additive-expression>
    <multiplicative-expression>
    <additive-expression> + <multiplicative-expression>
    <additive-expression> - <multiplicative-expression>

<shift-expression>
    <additive-expression>
    <shift-expression> << <additive-expression>
    <shift-expression> >> <additive-expression>

<relational-expression>
    <shift-expression>
    <relational-expression> < <shift-expression>
    <relational-expression> > <shift-expression>
    <relational-expression> <= <shift-expression>
    <relational-expression> >= <shift-expression>

<equality-expression>
    <relational-expression>
    <equality-expression> == <relational-expression>
    <equality-expression> != <relational-expression>

<and-expression>
    <equality-expression>
    <and-expression> & <equality-expression>
```

```
<exclusive-or-expression>
    <and-expression>
    <exclusive-or-expression> ^ <and-expression>

<inclusive-or-expression>
    <exclusive-or-expression>
    <inclusive-or-expression> | <exclusive-or-expression>

<logical-and-expression>
    <inclusive-or-expression>
    <logical-and-expression> && <inclusive-or-expression>

<logical-or-expression>
    <logical-and-expression>
    <logical-or-expression> || <logical-and-expression>

<conditional-expression>
    <logical-or-expression>
    <logical-or-expression> ? <expression> : <conditional-
expression>

<expression>
    <conditional-expression>
```



## 第 6 章 デバッガの環境設定

この章では、CodeWarrior デバッガの環境設定を説明します。各設定パネルのオプションとその動作を解説します。

### 環境設定の概要

環境設定パネルでデバッガの挙動を設定します。環境設定パネルには以下の二つのカテゴリがあります。

[MW Debug の環境設定パネル](#)

[デバッガのターゲット設定パネル](#)

### MW Debug の環境設定パネル

デバッガには、すべてのターゲットに共通の環境設定パネルと、特定のターゲットのためのパネルがあります。ターゲット特有の設定パネルについては、それぞれの『Targeting』マニュアルをご覧ください。共通の環境設定パネルは以下の通りです。

[Settings 環境設定パネル](#)

[Display 環境設定パネル](#)

[Symbolics 環境設定パネル](#)

[Program Control 環境設定パネル](#)

[Win32 Settings 環境設定パネル](#)

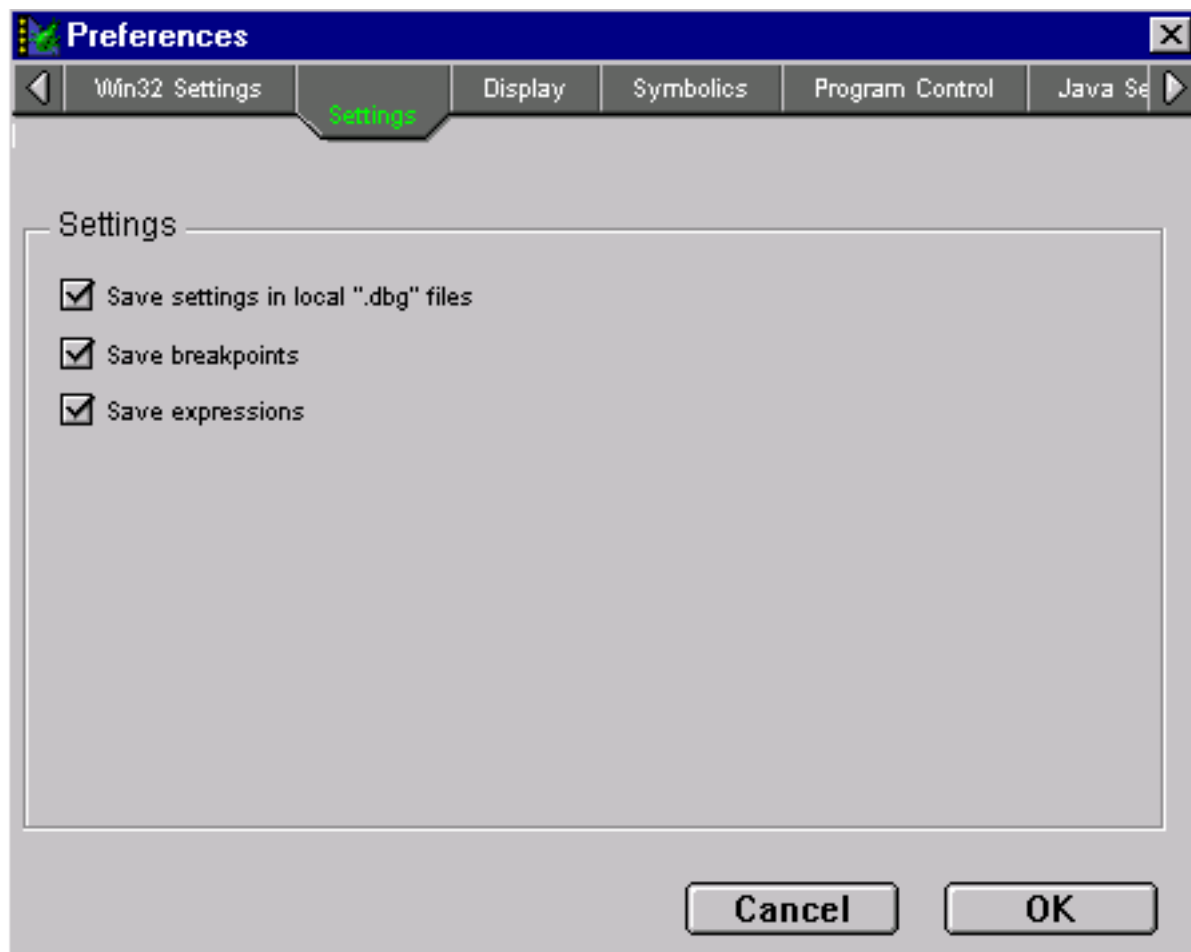
[Java Settings 環境設定パネル \(Windows\)](#)

[Runtime Settings 環境設定パネル \(Windows\)](#)

### Settings 環境設定パネル

Settings 環境設定パネルを [図 6.1](#) に示します。このパネルにはデバッガの設定を保存するオプションがあります。

図 6.1 Settings 環境設定パネル



#### Save settings in local ".dbg" files

ウィンドウのサイズと位置を .dbg ファイルに保管します。このファイルにはブレークポイントと評価式も含むことができます。デバッガが開くそれぞれのシンボルファイルに対して、このパネルで新しい .dbg ファイルを作成するか、または既存のファイルを修正して指定します。このオプションがオフの場合も、.dbg ファイルは作成されます。しかし、ウィンドウやその他のデータは保管されません。

#### Save breakpoints

『Save window settings in local ".dbg" files』がオンの場合、利用可能になります。ブレークポイントの設定をシンボルファイルの .dbg ファイルに保管します。オフの場合、ブレークポイントの設定は .dbg ファイルに保管されずに、捨てられます。

### Save expressions

『Save window settings in local ".dbg" files』がオンの場合、利用可能になります。Expression ウィンドウの内容をシンボルフайルの .dbg ファイルに保管します。オフの場合、Expression ウィンドウの内容は .dbg ファイルに保管されずに、捨てられます。

[『Expression ウィンドウ』\(p35\)](#) を参照してください。

### Display 環境設定パネル

Display 環境設定パネルを [図 6.2](#) に示します。デバッガウィンドウに表示するアイテムの保存に関するオプションがあります。

#### In variable panes, show variable types by default

新しい変数ウィンドウを開くときに変数型を表示します。この設定は .dbg ファイルに保管されます。

プロジェクトの .dbg ファイル内の設定は、このオプションより先に使われます。このオプションを設定する前に開かれた変数ウィンドウは、.dbg ファイル内の設定を使います。

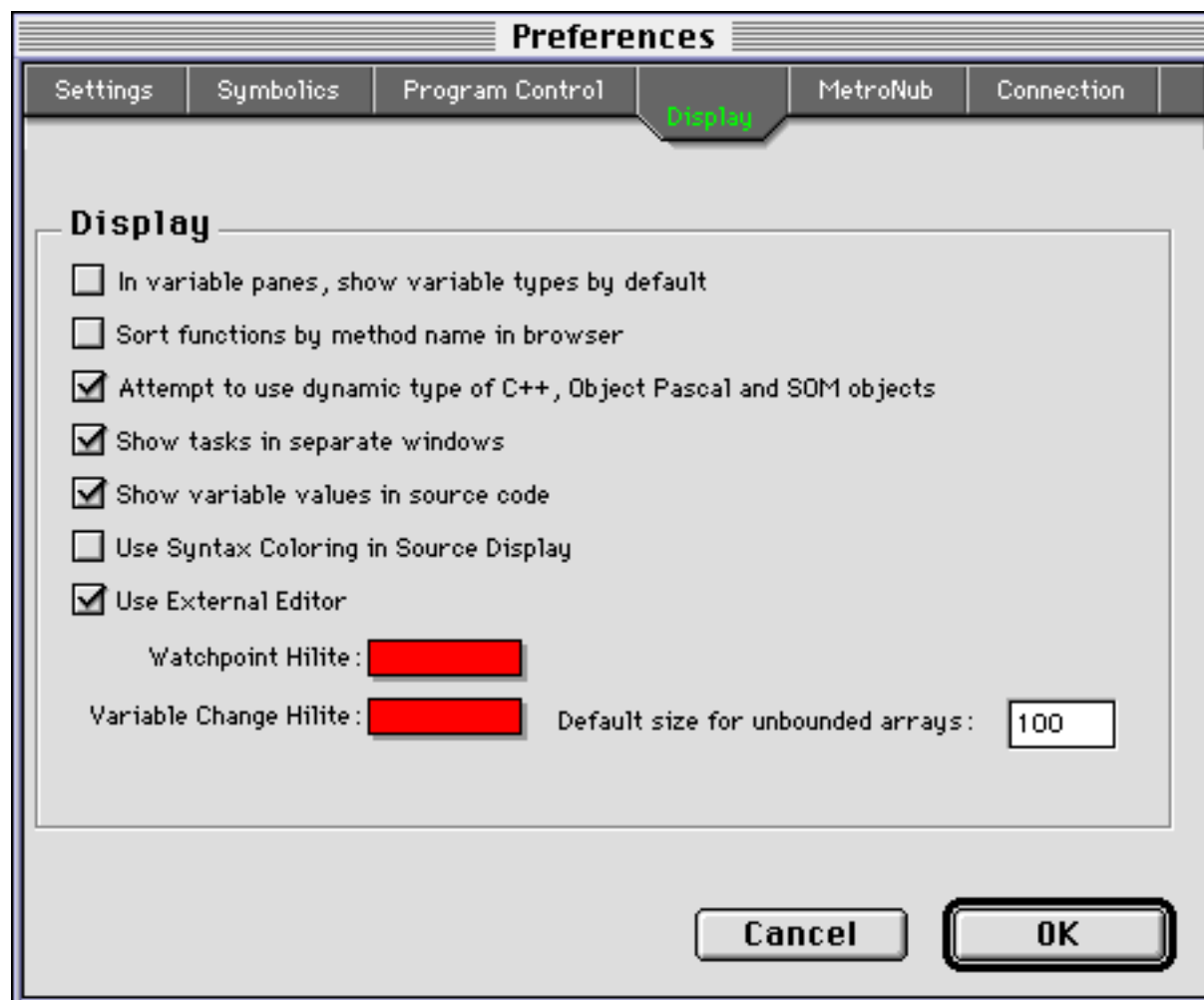
#### Sort functions by method name in browser

C++、Object Pascal、および Java の関数がブラウザウィンドウの関数欄でソートされる方法を変更します。このオプションがオフの場合、関数名は `className::methodName` の形式でクラス名を先、メソッド名を後にしてアルファベット順に並べられます。オンの場合、メソッド名だけをアルファベット順に並べます。ほとんどの C++、Object Pascal、および Java のソースファイルでは、一つのクラスのすべてのメソッドを一つのファイルに書きますので、このオプションは関数欄でメソッドを選択する際にキーボードからメソッド名を入力することで簡単に選択できるようになります。

#### Show variable values in source code

これをオンにした場合、ソースコード上でカーソルを変数名の上におくと変数の値を表示します。オフの場合、変数の値は表示されません。

図 6.2 Display 環境設定パネル



Attempt to use dynamic type of C++, Object Pascal objects and SOM objects

C++ または Object Pascal のオブジェクトの実行時の型を表示します。このオプションがオフの場合、オブジェクトのスタティック型のみを表示します。デバッガは、少なくとも一つの仮想関数を持つクラスのみをダイナミックな型として判別します。仮想基底クラスはサポートされません。

Show tasks in separate windows

タスクの表示方法を選択できます。このオプションをオンにすると、Process ウィンドウのタスクをダブルクリックするだけでコード表示するスタックロール欄が現れます。オフの場合、スタックロール欄の下部に Thread ポップアップメニューが現れます。タスクを同じスタックロールウィンドウに表示するか否かをこのメニューを使って選択します。



注意： このオプションが反映されるまで多少時間がかかります。デバッグ作業の初めから選択されていると、作業の間中は選択されたままです。デバッグ作業の途中でオプション設定を変更するためには、デバッグを中止してから再開しないと、設定の変更は反映されません。

#### Use Syntax Coloring in Source Display

ソースコードをシンタックスカラーで表示するかを設定します。このチェックボックスがオンの場合、ソースコードと関数の色が異なります（例えば、コメントは緑色で表示されます）。オフの場合、ソースコードはターゲットのデフォルトで表示されます。

#### Use External Editor

CodeWarrior 以外のエディタでソースコードを編集するかを設定します。

#### Watchpoint Hilite

デバッガが watchpoint を識別するための色を設定します。カラーフィールドをクリックすると標準のカラーピッカーダイアログが表示されます。デフォルトの色は赤です。

#### Variable Change Hilite

デバッガが変化した変数を識別するための色を設定します。カラーフィールドをクリックすると標準のカラーピッカーダイアログが表示されます。デフォルトの色は赤です。

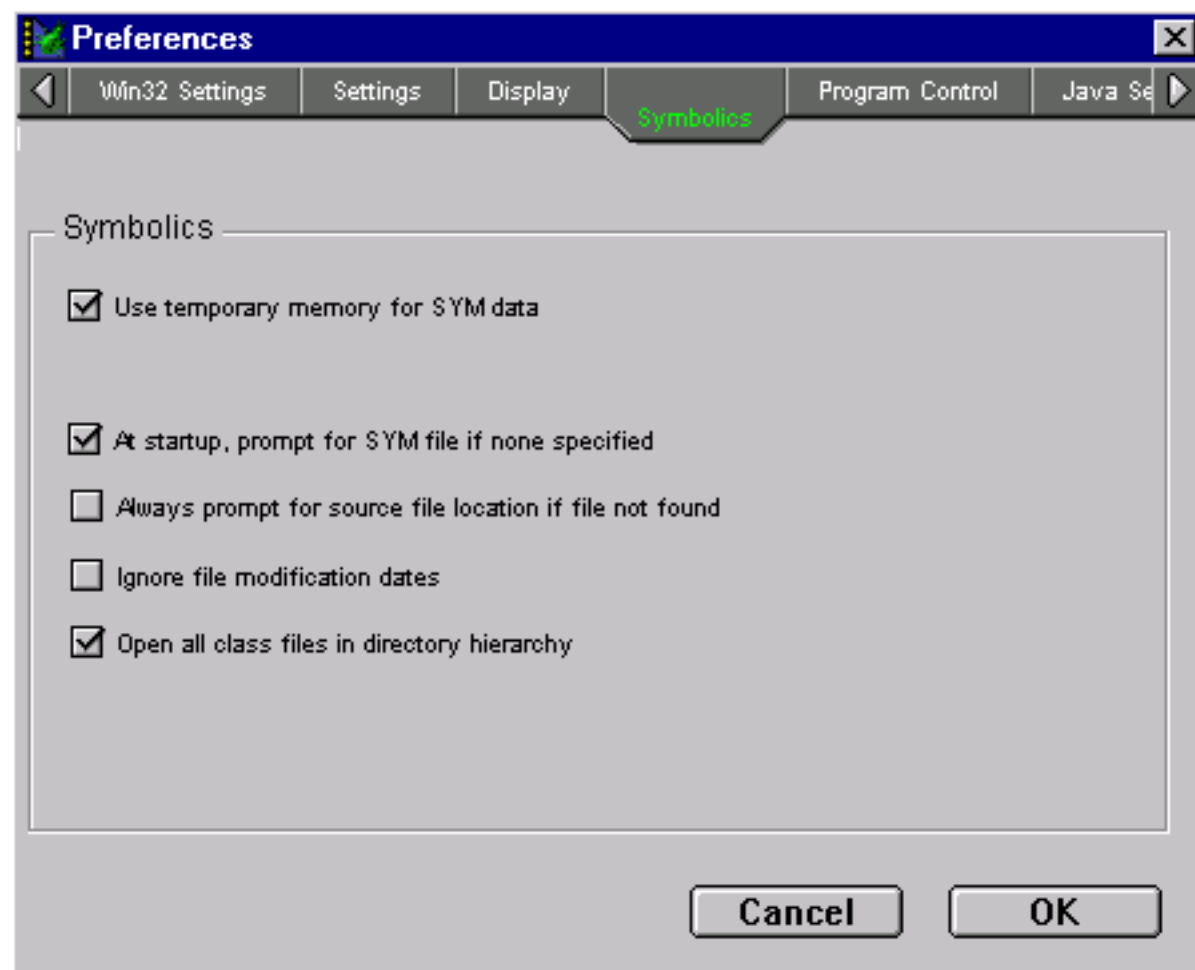
#### Default size for unbound arrays

サイズ情報が利用可能でないときに使う配列サイズを指定します。

## Symbolics 環境設定パネル

Symbolics 環境設定パネルを [図 6.3](#) に示します。シンボルファイル、Java クラス、zip ファイルの操作に関するオプションがあります。

図 6.3 Symbolics 環境設定パネル



#### Use temporary memory for SYM data

シンボルファイルのデータを保管するために一時メモリを使います。これにより、デバッガのメモリパーティションを小さくでき、実行するターゲットプログラムへの干渉を少なくすることができます。このオプションがオフの場合、デバッガが使うメモリが増え、ターゲットプログラムが使用できるメモリが少なくなります。

#### At startup, prompt for SYM file if none specified

デバッガだけを起動したときに開くシンボルファイルを要求するダイアログを出します。このオプションがオフの場合、シンボルファイルを指定するダイアログを表示しないでデバッガを起動できます。

#### Always prompt for source file location if file not found

デバッガがターゲットプログラムのソースファイルを見つけられないときに、ファイルの位置を指定するためのダイアログを表示します。通常デバッガは必要なファイルの位置を

覚えています。このオプションをオンにすると、以前にファイルの位置を記録している場合でも見つからないソースファイルの位置を指定するためのダイアログを表示します。

#### Ignore file modification dates

デバッガは、シンボルファイルが作成されたときからソースファイルの修正日時を追跡しています。修正日時が合わない場合、通常デバッガは、オブジェクトコードとソースコードに不一致がある可能性を提示する警告を出します。このオプションがオンの場合、この警告を表示しません。オフの場合、警告が表示されます。

#### Open all class files in directory hierarchy

このオプションをオンにすると、デバッガはディレクトリ内のすべてのクラスファイルとサブディレクトリを開いて、同じブラウザウィンドウ内に一緒に表示します。

ターゲット固有のオプションについては、『Targeting』マニュアルをご覧ください。

### Program Control 環境設定パネル

Program Control 環境設定パネルを [図 6.4](#) に示します。デバッグするプログラムのコントロールに関するオプションがあります。(単独デバッガの Program Control 設定パネルは IDE の統合デバッガでは Edit メニューの Preferences の Debugger カテゴリの Global Settings パネルに相当します。)

#### Automatically launch applications when SYM file opened

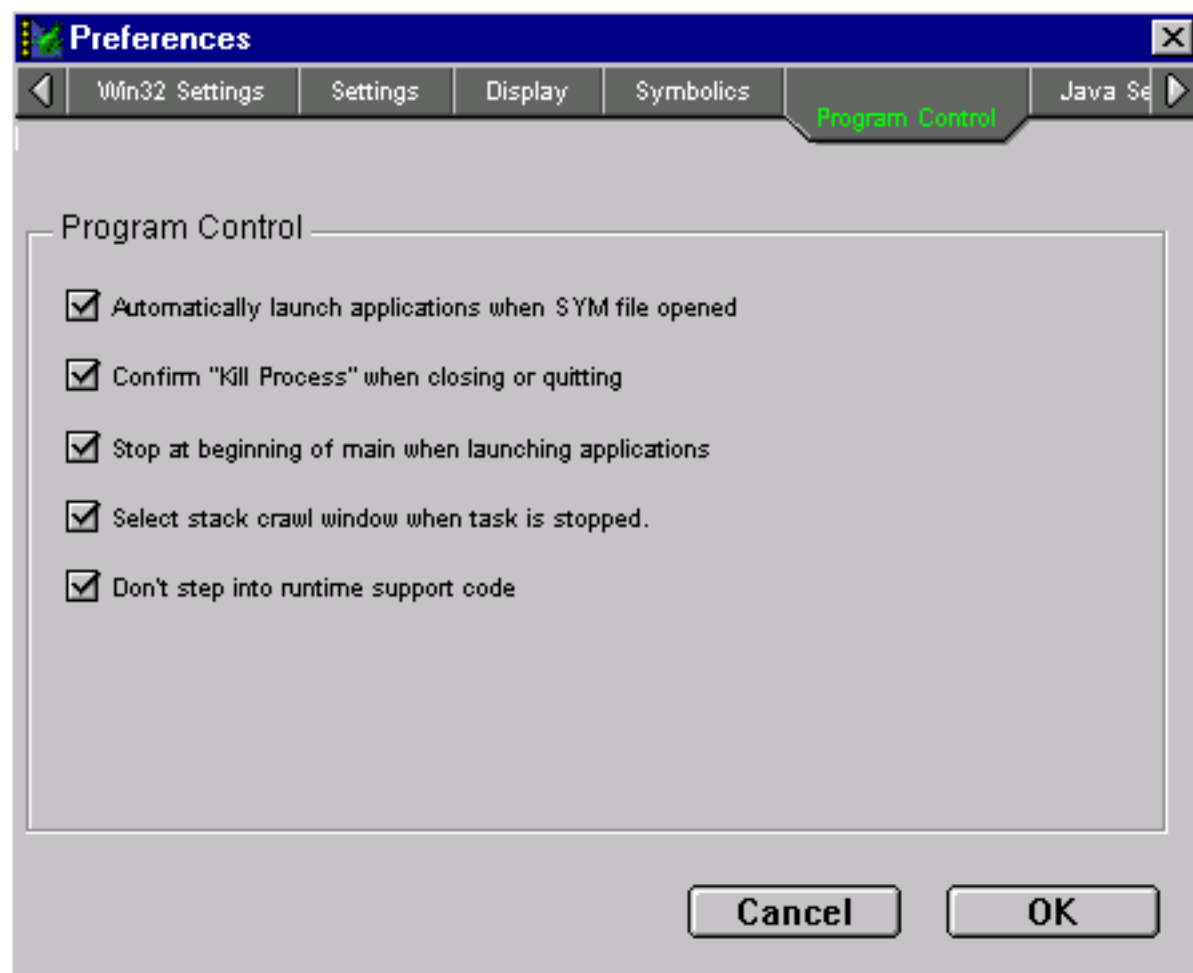
シンボルファイルを開いたときに、プログラムのメインエントリポイントに暗示的ブレークポイントを設定して、ターゲットプログラムを自動的に起動します。このオプションがオフの場合、シンボルファイルを開いたときにプログラムを起動しません。C++ のコンストラクタ関数などのように、メインルーチンの前に実行されるオブジェクトコードを検査できます。

Alt/Option キーを押しながらシンボルファイルを開くと、ターゲットプログラムは起動しません。

#### Confirm "Kill Process" when closing or quitting

ターゲットプログラムを終了したときに、プロセスを打ち切る前に確認のダイアログを表示します。

図 6.4 Program Control 環境設定パネル



#### Stop at program main when launching applications

通常アプリケーションのデバッグを始めるとき、デバッガは `main()` の最初の行で停止します。この後、このポイントを通過するために『Run』コマンドを選択する必要があります。このオプションをオフにすることにより、`main()` で停止しなくなります。ブレークポイントを設定した後に、このオプションを設定すると便利です。

#### Select stack crawl window when task is stopped

タスクが終了したときに自動的にスタッククロールウィンドウを最前面に表示します。このオプションがオフの場合、スタッククロールウィンドウの位置は変わりません。変数ウィンドウを開いていて、コードをステップ実行したときの変数の変化を見たいときに有効です。スタッククロールウィンドウがアクティブでないときでもコントロールできます。

図 6.5 Program Control 環境設定パネル ( Mac OS )



Don't step into runtime support code

C++ の static オブジェクトのコンストラクタ関数コードを、プログラムウィンドウ内に表示せずに、通常に実行します。

QC-aware ( Mac OS )

MW Debug で Onyx Technology の QC 機能拡張を使えるようにします。QC がエラーを報告すると、デバッガはエラーが発生したポイントでターゲットプログラムを停止して、アラートを表示します。QC エラーを報告した後、デバッガは QC を非活動状態にします。デバッグ開始前および QC のエラー報告の後は、QC ホットキーを使って QC を再度アクティブにしておきます。

このオプションがオフの場合、デバッガは QC エラー報告を無視して、QC の非活動化を行いません。

注意： 統合デバッガは 68K、PowerPC 両方の DebugStr() トラップをキャッチします。  
『QC-aware』 オプションは統合デバッガでは使用できません。

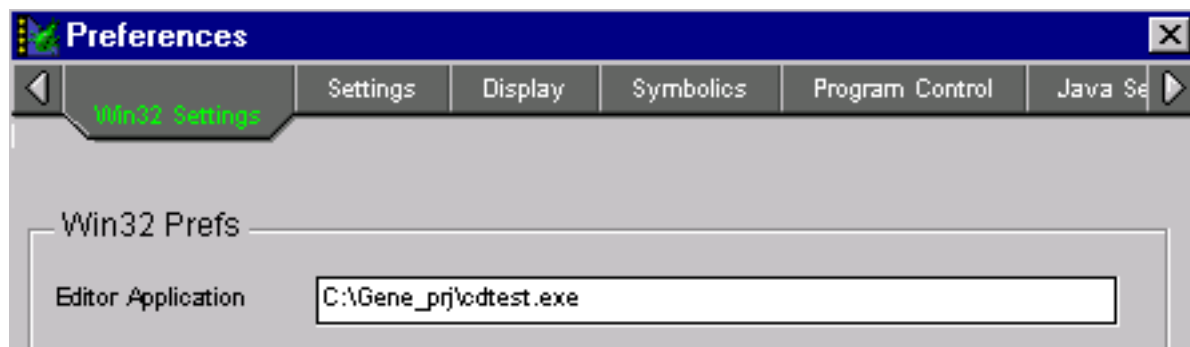
### Java Runtime ( Mac OS )

Java Applet をデバッグする際に、Metrowerks Java runtime、Apple MRJ、Internet Explorer のどれを使うかをこの項目で選択できます。詳細は『Targeting Java VM』をご覧ください。

### Win32 Settings 環境設定パネル

Win32 Settings 環境設定パネルを [図 6.6](#) に示します。このパネルでファイルの編集に使うデフォルトのエディタを設定します。デフォルトのエディタアプリケーションが指定されていない場合、Note Pad を使います。

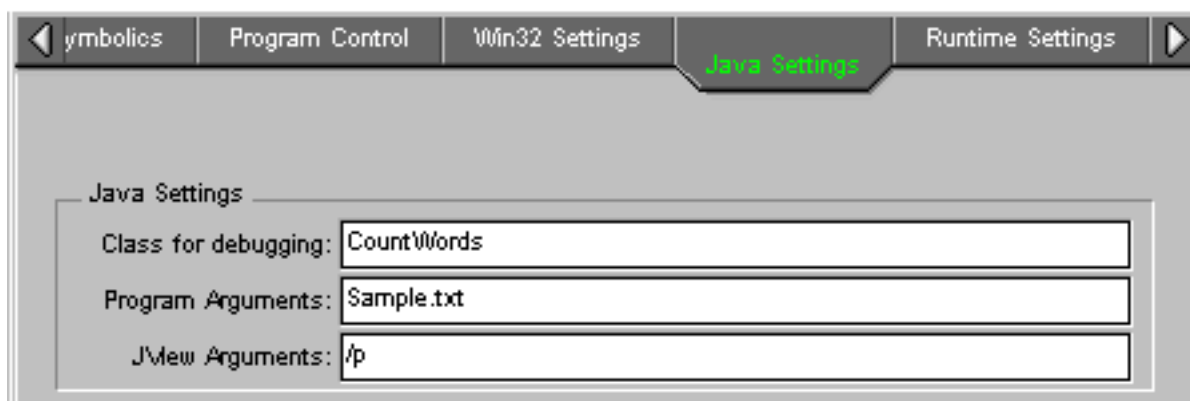
図 6.6 Win32 Settings 環境設定パネル



### Java Settings 環境設定パネル ( Windows )

Java Settings 環境設定パネル( [図 6.7](#) )には Java プログラムとアプレットのデバッグオプションがあります。

図 6.7 Java Settings 環境設定パネル



Class for debugging : デバッグしたいクラスファイルを指定します。

Program Arguments : Java アプリケーションのデバッグ時に、プロジェクトが使うコマンド行の引数を指定します。

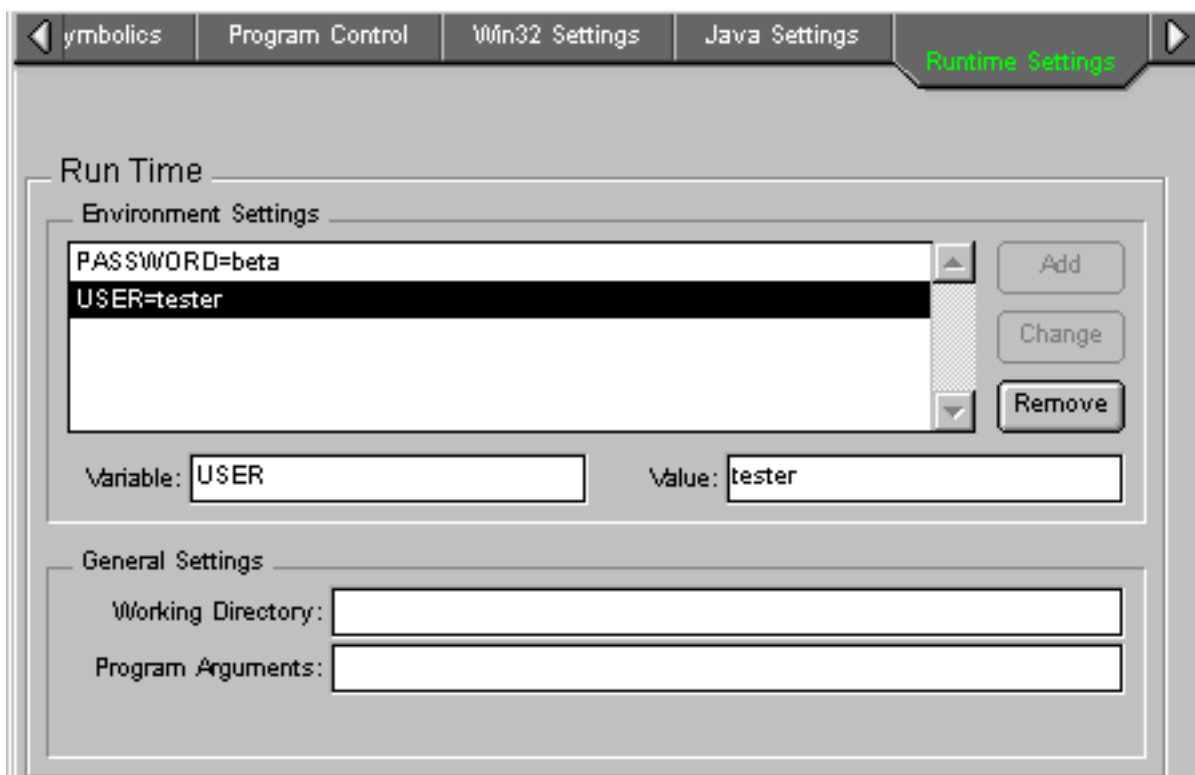
JView Arguments : プロジェクトのデバッグ時に、jview が必要とする引数があれば指定します。

Java プログラムとアプレットのデバッグの詳細は『Targeting Java』マニュアルを参照してください。

## Runtime Settings 環境設定パネル ( Windows )

Runtime Settings 環境設定パネル ( 図 6.8 ) には Windows の環境設定に関するオプションがあります。このパネルは Environment Settings と General Settings の二つの部分に分かれています。

図 6.8 Runtime Settings 環境設定パネル



Environment Settings 部分ではプログラムに `main()` の `envp` パラメータの一部として設定し、渡す環境変数を指定します。この変数は `getenv()` 呼び出しで利用できますが、ターゲットプログラム以外では使用できません。プログラム終了時にはこの設定は使用できません。例えば、サーバをログするプログラムでは、変数をユーザー ID とパスワードに使用することができます。

General Settings 部分には二つのフィールドがあります。

Working Directory：デバッグを行うワーキングディレクトリを指定します。指定がない場合、実行可能ファイルのあるディレクトリでデバッグを行います。

Program Arguments：プログラムの起動時に渡すコマンド行の引数を指定します。『Run』コマンドでプログラムが起動したときにこの引数が渡されます。

## デバッガのターゲット設定パネル

CodeWarrior の統合デバッガの設定パネルについて説明します。

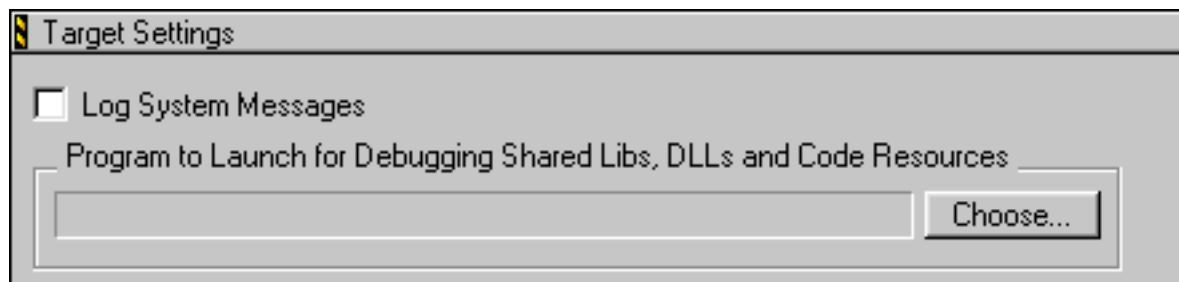
[Target Settings 設定パネル](#)

[x86 Exceptions 設定パネル \( Windows \)](#)

### Target Settings 設定パネル

Target Settings 設定パネル ( [図 6.9](#) ) は、ログの実行と共有ライブラリ、DLL、コードリソースをデバッグするときに起動するアプリケーションを指定します。

図 6.9 Target Settings 設定パネル



#### Log System Messages

このオプションがオンの場合、すべてのシステムメッセージをファイルにログします。オフの場合、ログファイルは生成されません。

#### Program to Launch for Debugging Shared Libs, DLLs and Code Resources

[ Choose ] ボタンをクリックして、共有ライブラリ、DLL、コードリソースをデバッグするときに起動するアプリケーションの名前を選択します。

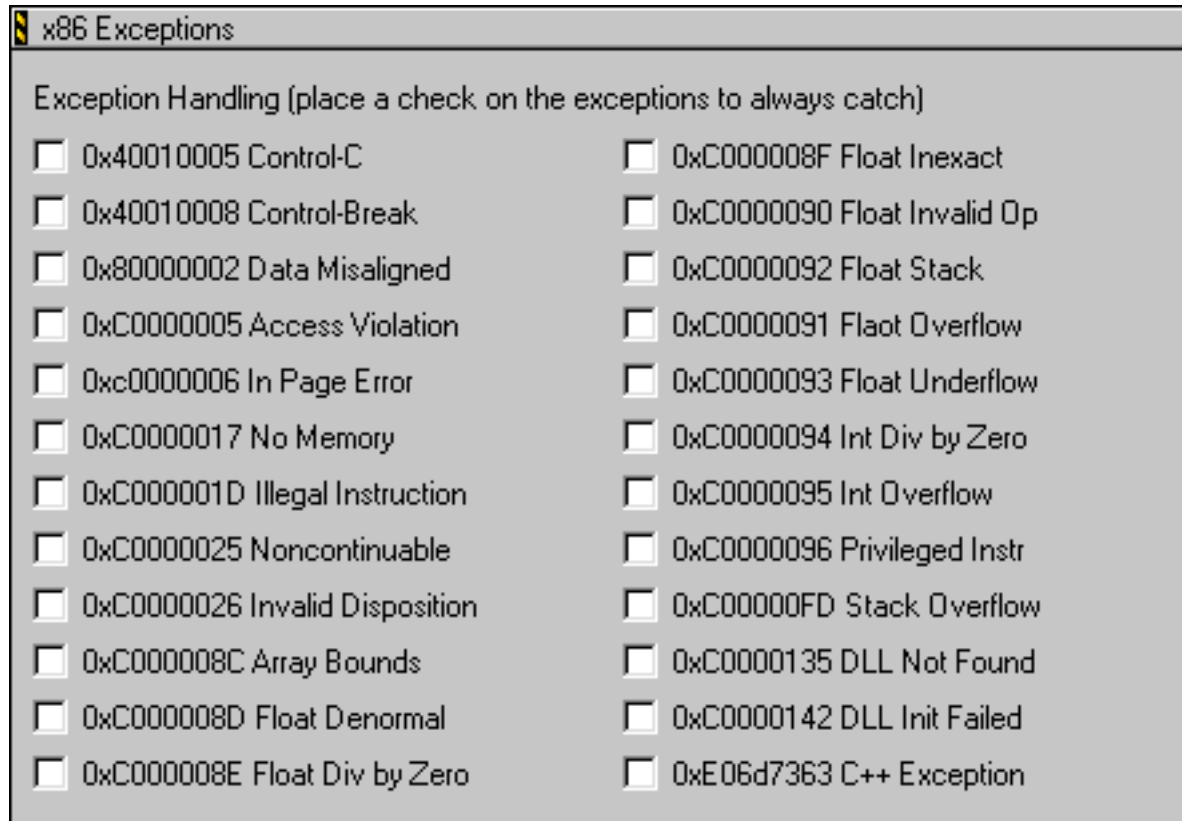
これはデバッグアプリケーションではありません。共有ライブラリ、DLL、コードリソースが対応するアプリケーションです。例えば、Photoshop のプラグインを作成した場合、ターゲットアプリケーションとして Photoshop を選択します。Debug を選択した場合、IDE はプラグインを作成し、プラグイン用のシンボル情報をロードします。次に Photoshop を起動してプラグインのデバッグを可能にします。



## x86 Exceptions 設定パネル ( Windows )

x86 Exceptions 設定パネル ( [図 6.10](#) ) で、統合デバッガでキャッチする例外を指定します。

図 6.10 x86 Exceptions 設定パネル







## 第 7 章 デバッガのメニュー

MW Debug のメニューについて説明します。

### デバッガメニューの概要

デバッガには、次の六つのメニューがあります。

[File メニュー](#)：シンボルファイルを開くとき、閉じるとき、ソースファイルを開くとき、デバッガを終了するときに使います。

[Edit メニュー](#)：標準の編集コマンドを実行するとき、デバッガ環境を設定するときに使います。

[Control メニュー](#)：コードの中で移動するとき、ローレベルデバッガに切り替えるときに使います。

[Data メニュー](#)：デバッガのデータの表示形式を選択するときに使います。

[Window メニュー](#)：デバッガのウィンドウを開くとき、また閉じるときに使います。

[Help メニュー \( Windows \)](#)：MW Debug についての情報を表示します。

[Apple メニュー \( Mac OS \)](#)：MW Debug についての情報を表示します。

---

注意： デバッガのメニューコマンドの位置が IDE の統合デバッガと異なるかもしれません。詳細は『IDE User Guide』をご覧ください。

---

### File メニュー

File メニューのコマンドは、ファイルを開く、閉じる、編集する、および保存するために使います。

#### Open

デバッグするために、既存のシンボルファイルを開きます。シンボルファイルを選択するための標準の File ダイアログが表示されます。シンボルファイルは、ターゲットプログラム（デバッグしたいプログラム）と同じフォルダに置く必要があります。

シンボルファイルを選択すると、デバッガがシンボルファイルとターゲットプログラムをメモリにロードし、プログラムのメインエントリポイントに暗示的なブレークポイントを設定してプログラムを起動します。プログラムが最初のブレークポイントまで実行されると、プログラムは停止して、コントロールがデバッガに戻ります。

『Open』 コマンドは、Java クラスや zip ファイルを開くときにも使います。デバッガは、これらのファイルからシンボルを読み取り、シンボルファイルと同じように処理します。詳しくは、『Targeting Java Manual』をご覧ください。

参照：ターゲット用のシンボル情報の生成については、[『デバッグの準備』\(p15\)](#) もご覧ください。

---

注意： 同時に複数のプログラムを開いてデバッグできます。

---

## Close

アクティブなウィンドウを閉じます。

『Confirm "Kill Process" when closing or quitting』 オプションがオフの場合、プログラムウィンドウを閉じると実行中のプログラムも終了します。『Run』 コマンドはプログラムウィンドウを再度開いて、プログラムを最初から再実行します。このオプションがオンの場合、ダイアログが表示され、プログラムを終了するか、プログラムウィンドウを閉じた後も実行し続けるか、または『Close』 コマンドをキャンセルするかを選択できます。

[『Confirm "Kill Process" when closing or quitting』 \(p99\)](#) もご覧ください。

## Edit filename

指定されたソースファイルを CodeWarrior IDE エディタで開きます。CodeWarrior IDE は既に実行されている必要があります。デバッガは IDE を自動的に起動しません。

指定されたソースファイルを、Win32 Settings 環境設定パネルで選択されたデフォルトのエディタを使って開きます。デフォルトのエディタアプリケーションを指定していない場合、NotePad を使ってファイルを開きます。

[『Win32 Settings 環境設定パネル』 \(p102\)](#) もご覧ください。

## Save

Log ウィンドウの内容を、現在のファイル名でディスクに保存します。このコマンドは、Log ウィンドウがアクティブなときだけ有効です。

## Save As

Log ウィンドウの内容を違う名前のファイルに保存するための、標準の File ダイアログを表示します。新しい名前は Log ウィンドウに関連付けられ、後で『Save』コマンドを使ったときに、この名前が使われます。このコマンドは、Log ウィンドウがアクティブなときだけ有効です。

## Save A Copy As

Log ウィンドウの内容を違う名前のファイルに保存するための、標準の File ダイアログを表示します。Log ウィンドウに関連付けられた古い名前はそのまま残し、後で『Save』コ

マンドを使ったときには、新しい名前ではなく古い名前が使われます。このコマンドは、Log ウィンドウがアクティブなときだけ有効です。

### Save Settings

設定パネルの『Save settings in local ".dbg" files』オプションがオンの場合、プログラムウィンドウおよびブラウザウィンドウの現在の設定を保存します。同様に『Save Breakpoints』や『Save Expressions』がオンの場合、それぞれブレークポイントおよび評価式を保存します。

[『Settings 環境設定パネル』\( p93 \)](#) もご覧ください。

### Quit

デバッガを終了します。

環境設定パネルの『Confirm "Kill Process" when closing or quitting』がオフの場合、デバッガを終了したときにすべてのプログラムを終了します。このオプションがオンの場合、デバッガを終了した後もプロセスの実行を継続させたり、『Quit』コマンドをキャンセルしたりすることができます。

[『Confirm "Kill Process" when closing or quitting』\( p99 \)](#) もご覧ください。

## Edit メニュー

Edit メニューのコマンドは、デバッガで表示されている変数の値や評価式に対して使います。デバッガではソースコードの編集はできません。

ソース欄に表示されているソースコードの編集方法については、[『Edit filename』\( p108 \)](#) をご覧ください。

### Undo

最新の『Cut』、『Copy』、『Paste』、または『Clear』操作を取り消すには、『Undo』を選択してください。

### Cut

選択したテキストを削除し、クリップボードにコピーするには、『Cut』を選択してください。ソース欄ではテキストをカットできません。

### Copy

選択したテキストをクリップボードにコピーするには、『Copy』を選択してください。ソース欄または Log ウィンドウのテキストをコピーすることができます。

### Paste

クリップボードにあるテキストをアクティブウィンドウにペーストするには、『Paste』を選択してください。ソース欄にはペーストできません。

### Clear

選択したテキストをクリップボードにコピーしないで削除するには、『Clear』を選択してください。ソース欄ではテキストを削除できません。

### Select All

アクティブウィンドウのすべてのテキストを選択するには、『Select All』を選択してください。変数の値または評価式を編集しているとき、または Log ウィンドウ内でテキストを選択できます。

### Find

プログラムのソース欄またはブラウザウィンドウ内のテキストを検索するための検索ダイアログを表示します。検索は、現在選択されている部分または挿入位置から始まり、ファイルの最後まで行います。

[『Find ダイアログの使い方』\(p62\)](#) もご覧ください。

### Find Next

現在の選択位置または挿入位置から始めて、最後の検索を繰り返します。

### Find Selection

ソース欄で現在選択されているテキストが次に現れる場所を検索します。このコマンドは、選択されているテキストがない場合や、挿入点だけの場合は使えません。

---

ヒント： Shift + Ctrl/Command+G ( Find Next ) または Shift + Ctrl/Command + H ( Find Selection ) を押すと、検索の方向を反対にできます。

---

[『Find ダイアログの使い方』\(p62\)](#) もご覧ください。

### Preferences

デバッガの環境を設定するには、『Preferences』を選択してください。デバッガのさまざまな動作を設定するためのダイアログが表示されます。Preferences ダイアログについては、[『環境設定の概要』\(p93\)](#) をご覧ください。

## Control メニュー

Control メニューには、プログラム実行を管理するコマンドがあります。

### Run

ターゲットプログラムを実行します。カレントステートメント矢印の位置から実行され、ブレークポイントに達するまで、または『Stop』または『Kill』コマンドを使うまで実行されます。

[『コードを実行』\(p51\)](#) もご覧ください。

## Stop

ターゲットプログラムの実行を一時停止しデバッガに戻るには、『Stop』を選択してください。実行されているプログラムを『Stop』で一時停止すると、ローカル変数の現在の値を示すプログラムウィンドウが表示されます。カレントステートメント矢印は、次に実行されるステートメントに配置されます。Control メニューの『Stop』は薄く表示され、プログラムウィンドウのソース欄にメッセージが表示されます。

一時停止したプログラムの実行を続けるためには、以下の方法があります。

『Run』 コマンドを選択します。カレントステートメント矢印の位置から実行が再開されます。

Control メニューの『Step Over』、『Step Into』、『Step Out』を使って、ターゲットプログラムのステートメントを一つずつステップ実行します。

---

注意：『Stop』 コマンドはターゲットのオペレーティングシステムに依存します。詳細は各『Targeting』 マニュアルを参照してください。

---

[『実行を停止する』\(p55\)](#) もご覧ください。

## Kill

ターゲットプログラムの実行を停止しデバッガに戻るには、『Kill』を選択してください。ブレークポイントまたは『Stop』 コマンドを使うと、停止した位置からプログラム実行を再開できますが、『Kill』 コマンドの場合は『Run』 コマンドを使ってプログラムのメインエントリポイントから再度実行する必要があります。

[『プログラムを終了する』\(p56\)](#) もご覧ください。

## Step Over

関数コールをステップオーバーして、一つのステートメントだけを実行するには、『Step Over』を選択してください。カレントステートメント矢印が指しているステートメントを実行した後、デバッガに戻ります。デバッガが関数コールを実行するとき、その関数のソースコードはプログラムウィンドウに表示されません。つまり『Step Over』は深いコールチェーンには入りません。しかし、関数の実行が終わったときは『Step Over』は関数の呼び出し元に戻って実行を続けます。

[『一行ずつ実行する』\(p52\)](#) もご覧ください。

## Step Into

関数コールにステップインして一つのステートメントだけを実行するには、『Step Into』を選択してください。カレントステートメント矢印が指しているステートメントを実行した後、デバッガに戻ります。『Step Over』と違い、『Step Into』は関数コールを追い、プログラ

ムウィンドウのソース欄にコールされた関数のソースを表示します。ステップインした関数の名前がプログラムウィンドウのコールチェーン欄に表示されます。

[『ルーチンの中へ入る』\(p53\)](#) もご覧ください。

#### Step Out

現行の関数の残りの部分を実行した後に呼び出し元に戻るには、『Step Out』を選択してください。『Step Over』や『Step Into』と違い、『Step Out』はカレントステートメント矢印が指しているステートメントからプログラムを実行し、関数がその呼び出し元に戻るときにデバッガに戻ります。

[『ルーチンの中から呼び出し元に戻る』\(p53\)](#) もご覧ください。

---

ヒント： オペレーティングシステムルーチンなどのように、デバッグ情報が含まれていない関数は、プログラムウィンドウのソース欄にアセンブリ言語が表示されます。デバッグ情報が含まれていない関数を実行し、呼び出し元に戻るには、『Step Out』を使ってください。

---

#### Clear All Breakpoints

ターゲットプログラムのソースファイルのブレイクポイントをすべて消去するには、『Clear All Breakpoints』を選択してください。

#### Break on C++ exception

この設定が選択されているとき、C++ の例外が発生する度にデバッガは `__throw()` 関数でブレイクします。C++ 例外のデバッグに関しては、それぞれの『Targeting』マニュアルをご覧ください。

#### Switch to Monitor ( Mac OS )

Macintosh ROM モニタプログラムまたはローレベルデバッガ( MacsBug など )がインストールされている場合に、コントロールをローレベルデバッガへ切り替えるには『Switch to Monitor』を選択してください。

---

注意： 逆方向に切り替えるための、一対の MacsBug のマクロファイルが、CodeWarrior デバッガとともに提供されています。ResEdit などを使って、Debugger Prefs ファイルにこれらのマクロを加えると、MacsBug デバッガでから CodeWarrior へ切り替えることができます。68K コードの場合は cw、Power Macintosh コードの場合は cwp と入力すると切り替わります。

---

## Data メニュー

Data メニューは、デバッガでデータ値を表示する方法をコントロールします。



## Show Types

アクティブな変数欄または変数ウィンドウに、ローカル変数やグローバル変数のデータ型を表示するには、『Show Types』を選択してください。

## Expand

選択した構造体変数の中の C メンバ変数、C++ データメンバ変数、Pascal フィールド変数、または Java フィールド変数を表示します。または選択したポインタまたはハンドルの参照を取り消します。

## Collapse All

C メンバ変数、C++ データメンバ変数、Pascal フィールド変数、Java フィールド変数、またはポインタやハンドルの参照を隠します。

## New Expression

Expression ウィンドウに新しい項目を作成し、新しい評価式を入力するためのプロンプトを表示します。ソースコードやその他のウィンドウや変数欄からドラッグ & ドロップで評価式を Expression ウィンドウへコピーすることもできます。評価式を選択後、Data メニューの『Copy To Expression』を選択することもできます。

[『Expression ウィンドウ』\(p35\)](#) もご覧ください。

## Open Variable Window

選択した変数を表示するために別のウィンドウを作成するには『Open Variable Window』を選択してください。このコマンドは、大きな構造体変数 (Pascal 言語の RECORD や C/C++ 言語の struct) をモニタするときに便利です。

[『変数ウィンドウ』\(p37\)](#) もご覧ください。

## Open Array Window

選択した配列を表示するために別のウィンドウを作成するには『Open Array Window』を選択してください。このコマンドは、配列の値をモニタするときに便利です。

[『配列ウィンドウ』\(p38\)](#) もご覧ください。

## Copy to Expression

アクティブな欄で選択した変数を Expression ウィンドウにコピーするには、『Copy to Expression』を選択してください。変数をソースコードまたはその他のウィンドウまたは欄から Expression ウィンドウにドラッグすることもできます。

[『Expression ウィンドウ』\(p35\)](#) もご覧ください。

### Set/Clear Watchpoint

選択した変数やメモリの範囲に対して Watchpoint を設定または消去するには、このコマンドを選択してください。メモリウィンドウで変数やメモリの範囲を選択したり、変数ウィンドウで変数を選択して Watchpoint を指定することができます。既に Watchpoint があるとき、このコマンドは『Clear Watchpoint』になります。

[『ブレークポイントを設定』\(p64\)](#) および [『ウォッチポイントを消去』\(p70\)](#) もご覧ください。

### Clear Current Watchpoint

プログラムがちょうど停止したところのウォッチポイントを消去します。

[『ウォッチポイントを消去』\(p70\)](#) もご覧ください。

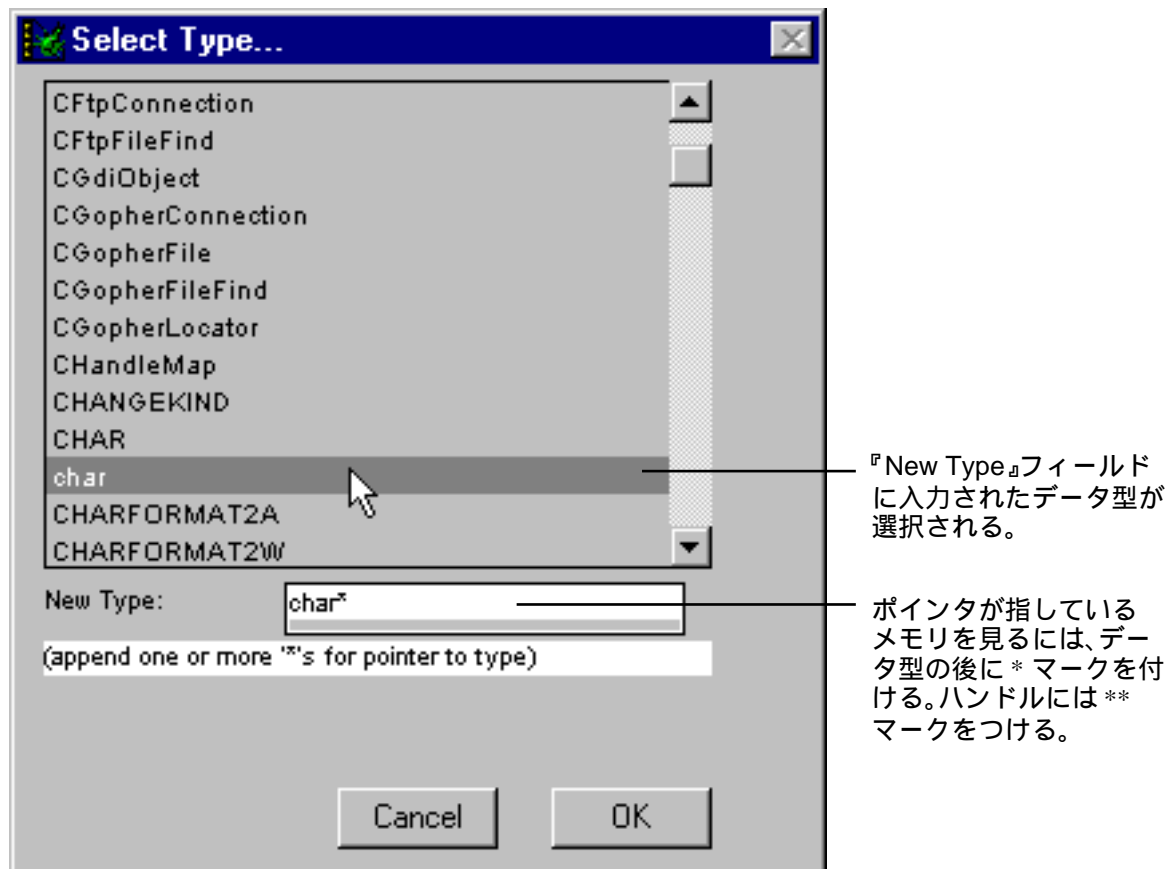
### View As

選択した変数を指定したデータ型で表示するには、『View as』を選択してください。このコマンドを選択すると、プログラムウィンドウのローカル変数欄、ブラウザウィンドウのグローバル変数欄、または変数ウィンドウに表示されている変数に適用できます。

メモリ変数は、どのようなデータ型としても表示できます。新しいデータ型が変数の元の型より小さい場合、余ったデータは無視されます。新しいデータ型が変数の元の型より大きい場合、必要なメモリ内容を読みます。ただし、レジスタ変数の場合にはレジスタと同じサイズの型としてだけ見ることができます。

『View as』を選択すると、プロジェクトで定義されているすべてのデータ型を一覧表示したダイアログが表示されます ([図 7.1](#))。データ型を選択すると、その型が『New Type』フィールドに表示されます。変数をポインタとして解釈したい場合は星印(\*)を、ハンドルとして解釈したい場合は星印を二つ(\*\*)付けます。変数を選択したデータ型として見るには、[OK] をクリックしてください。

図 7.1 View As ダイアログ



[『データを異なる型で表示する』\(p75\)](#)、[『メモリダンプを表示する』\(p78\)](#) および [『アドレスを指定してメモリを表示する』\(p79\)](#) をご覧ください。

#### View Memory As

選択した変数が使用しているメモリ、または選択したレジスタが指しているメモリを表示するには、『View Memory as』を選択してください。『View as』ダイアログで指定した型を配列としてメモリの内容を表示する配列ウィンドウを開きます。

[『配列ウィンドウ』\(p38\)](#)、および [『アドレスを指定してメモリを表示する』\(p79\)](#) もご覧ください。

#### View Memory

16 進数と ASCII 文字表示でメモリ内容を表示するには、『View Memory』を選択してください。このコマンドは現在選択されている項目または評価式のアドレスから始まるメモリ内容をメモリウィンドウに表示します。

[『メモリウィンドウ』\(p39\)](#) もご覧ください。

#### Default

選択した変数をその変数の型に基づいたデフォルトのフォーマットで表示します。

#### Signed Decimal

選択した変数を符号付き 10 進数の値として表示するには、『Signed Decimal』を選択してください。

[『データを異なるフォーマットで表示する』\(p74\)](#) もご覧ください。

#### Unsigned Decimal

選択した変数を符号なし 10 進数の値として表示するには、『Unsigned Decimal』を選択してください。

#### Hexadecimal

選択した変数を 16 進数の値として表示するには、『Hexadecimal』を選択してください。

#### Character

選択した変数を文字として見るには、『Character』を選択してください。

デバッガは、表示できない文字を示すために、ANSI C エスケープシーケンスを使用します。エスケープシーケンスは、\ (バックスラッシュ) に続けて 8 進数値、またはあらかじめ定義されているエスケープシーケンスを使用します。例えば、文字コード 29 は、'\35' と表示されます (35 は、29 を 8 進数として表現したもの)。タブ文字は、'\t' と表示されます。

#### C String

選択した変数を C 文字列として表示するには、『C String』を選択してください。C 文字列は、NULL 文字 ('\0') で終わる ASCII 文字のシーケンスです。C 文字列は、NULL 文字なしで表示されます。

表示できない文字については、[『Character』\(p116\)](#) もご覧ください。

#### Pascal String

選択した変数を Pascal 文字列として見るには、『Pascal String』を選択してください。Pascal 文字列は、第 1 バイト目に文字列の文字数を含み、それに続いて文字列そのものが入っています。第 1 バイト目は表示されません。

#### Floating Point

選択した変数を浮動小数点として表示するには、『Floating Point』を選択してください。

## Enumeration

選択した変数を enum 型として見るには、『Enumeration』を選択してください。typedef で定義された C/C++ enum 変数に対しては、コンパイラが提供するシンボル名を表示します。char、short、int、または long 変数に対するシンボル値は表示されません。

---

注意： enum 変数を編集するときは、10 進数で入力しなければなりません。

---

## Fixed

選択した変数を固定小数点として見るには、『Fixed』を選択してください。固定小数点変数は、シンボルファイルに 32 ビット整数として保管されているため、最初はこの形式で表示されます。この変数を固定小数点変数としてフォーマットするには、変数を選択し、『Fixed』を選択します。32 ビット長のデータは何でも固定小数点値としてフォーマットできます。

## Fract

選択した変数を Fract 型として見るには、『Fract』を選択してください。Fract 型変数は、固定小数点型変数と同じように働きます。シンボルファイルには 32 ビット整数で保管されているため、最初は 32 ビット整数として表示されます。『Fixed』コマンドと同様に、『Fract』コマンドを使って変数をフォーマットし、編集できます。32 ビット長のデータはすべて Fract 型としてフォーマットできます。

# Window メニュー

Window メニューには、デバッガのウィンドウを表示したり隠すためのコマンドがあります。また、画面上で現在開いているすべてのウィンドウのリストもあります。

## Show/Hide Toolbar ( Mac OS )

フローティングツールバーを表示します。表示されているときは非表示にします。ツールバーが画面上に表示されているか否かにより、このコマンドは『Show Toolbar』と『Hide Toolbar』で切り替わります。

## Show/Hide Processes

プロセスウィンドウを表示します。表示されているときは非表示にします。プロセスウィンドウが画面上に表示されているか否かにより、このコマンドは『Show Processes』と『Hide Processes』で切り替わります。

[『プロセスウィンドウ』\(p43\)](#) もご覧ください。

### Show/Hide Expressions

Expression ウィンドウを表示します。表示されているときは非表示にします。Expression ウィンドウが画面上に表示されているか否かにより、このコマンドは『Show Expressions』と『Hide Expressions』で切り替わります。

[『Expression ウィンドウ』\(p35\)](#) もご覧ください。

### Show/Hide Breakpoints

ブレークポイントウィンドウを表示します。表示されているときは非表示にします。ブレークポイントウィンドウが画面上に表示されているか否かにより、このコマンドは『Show Breakpoints』と『Hide Breakpoints』で切り替わります。

[『Breakpoint ウィンドウ』\(p35\)](#) もご覧ください。

### Show/Hide Watchpoints

Watchpoints ウィンドウを表示します。表示されているときは非表示にします。Watchpoints ウィンドウが画面上に表示されているか否かにより、このコマンドは『Show Watchpoints』と『Hide Watchpoints』で切り替わります。

[『Watchpoint ウィンドウ』\(p36\)](#) もご覧ください。

### Close All Variable Windows

開いているすべての変数ウィンドウおよび配列ウィンドウを閉じます。変数ウィンドウおよび配列ウィンドウが一つも開いていない場合、このコマンドは薄く表示され選択できません。

### Show/Hide Registers

レジスタウィンドウを表示します。表示されているときは非表示にします。レジスタウィンドウが画面上に表示されているか否かにより、このコマンドは『Show Registers』と『Hide Registers』で切り替わります。

[『レジスタウィンドウ』\(p41\)](#) もご覧ください。

### Show/Hide FPU Registers

FPU レジスタウィンドウを表示します。表示されているときは非表示にします。FPU レジスタウィンドウが画面上に表示されているか否かにより、このコマンドは『Show FPU Registers』と『Hide FPU Registers』で切り替わります (FPU を持たないターゲットでは、FPU レジスタは利用できません)。

[『レジスタウィンドウ』\(p41\)](#) もご覧ください。

#### その他の Window メニューの項目

その他の Window メニューの項目は、画面上に開いているウィンドウによって異なります。アクティブウィンドウにはチェックマークが付きます。他のウィンドウをアクティブウィンドウにするときは、以下の方法のいずれかを使います。

アクティブにしたいウィンドウをクリックする

アクティブにしたいウィンドウを Window メニューで選択する

アクティブにしたいウィンドウの Window メニューに表示されているキーを押す

### Help メニュー ( Windows )

Help メニューには、CodeWarrior デバッガのヘルプファイルにアクセスするためのコマンドがあります。『About Metrowerks Debugger』は、アプリケーションの作成者および著作権についての情報を表示します。

### Apple メニュー ( Mac OS )

Apple メニューには、『About Metrowerks Debugger』があります。このコマンドを選択すると、アプリケーションの作成者および著作権についての情報が表示されます。







## 第 8 章 トラブルシューティング

この章は、CodeWarrior デバッガについてよく質問される内容およびその解決方法について説明します。デバッガを使っているときに何らかの問題に直面したときは、まずこの章をご覧ください。同じような問題が他にも発生しており、簡単な解決方法が見つかることもあります。

### トラブルシューティングの概要

デバッガの使用中に問題が発生した場合、まずデバッガのプリファレンスファイルを削除してください。ファイル名は MWDebug.prf で、Windows ディレクトリの Metrowerks ディレクトリにあります。

この章では、プログラムのデバッグ中に発生したさまざまな問題について解説します。解決方法も記してあります。これ以外の問題に遭遇された方は Metrowerks のテクニカルサポートまでご連絡ください。

ここでは以下の内容について説明します。

[一般的な問題](#)

[デバッガ起動時の問題](#)

[デバッガ実行時の問題 / クラッシュ](#)

[ブレークポイントの問題](#)

[変数の問題](#)

[ソースファイルの問題](#)

### 一般的な問題

ここで紹介する解決法が役に立たない場合もあるかもしれません。そんなときは Metrowerks のテクニカルサポートへ連絡する前に、以下の操作を試してみてください。

プロジェクトにある再生成されたファイルとデータ (.(x)symbolics, .dbg ファイル、プリファレンス、バイナリを含む) を除去してください (CodeWarrior IDE で『Remove Binaries and Compact』を選択してください。詳細は『CodeWarrior IDE User Guide』を参照してください)。

ハードディスクに新しいバージョンの IDE とデバッガをインストールしてください。

機能拡張ファイルが衝突していないか確認してください。

できるだけ多くの機能拡張ファイルをオフにして、例題を実行してください。

## デバッガ起動時の問題

ここではデバッガの起動時に発生する問題について説明します。

### デバッガが起動しない

#### 問題

『Enable Debugger』が選択されているのに、アプリケーションを実行してもデバッガが起動しない。

CodeWarrior の Project メニューの『Run』または『Debug』コマンドが薄く表示され、選択できない。

#### 背景

CodeWarrior IDE からデバッガを自動的に起動できるのは、プロジェクトがアプリケーションである場合のみです。またデバッガは CodeWarrior IDE と同じフォルダになくてもなりません。

#### 解決方法

プロジェクトでアプリケーションを生成していることを確認してください。『Run』コマンドはアプリケーションを作成している場合のみ使用可能です。

CodeWarrior デバッガが CodeWarrior IDE と同じフォルダにあるかを確認してください。

デバッガアイコンをダブルクリックして直接起動してください。

[『MW Debug を直接起動』\(p19\)](#) を参照してください。

### デバッグできない

#### 問題

『Run』コマンドを選択しても何も起こらない。

#### 背景

デバッガを実行するためには設定を正しく行わなくてはなりません。またコードが実際に何を行っているかを確認してください。

#### 解決方法

デバッガが有効になっているかを確認してください。

デバッガのリリースノートを読んで、サードパーティのソフトウェアとの互換性を確認してください。

[『デバッグのためにターゲットを準備』\(p15\)](#) を参照してください。

## 起動時のエラー (Mac OS)

### 問題

デバッグの開始時に『error -27』、終了時に『error -619』が発生する。

### 背景

デバッガと互換性のない、古いバージョンの RamDoublor を使用しています。

### 解決方法

RamDoublor のバージョンを 1.5.2 へアップグレードしてください。

## 起動が遅い

### 問題

大きなプロジェクトでデバッガを実行すると、デバッガが起動するまで数分かかり、ハードディスクから大きな音が聞こえる。

### 背景

デバッガはテンポラリメモリにシンボル情報を格納します。仮想メモリを使用している場合、デバッガがテンポラリメモリを使わないようにしてください。

### 解決方法

デバッガのプリファレンスダイアログで、『Use temporary memory for SYM data』をオフにしてください。デバッガのパーティションサイズをシンボルファイルのサイズにまで増やしてください。

## デバッガ実行時の問題 / クラッシュ

ここではデバッガの実行中に起こる問題について説明します。

### プロジェクトをデバッガなしで実行するとクラッシュする

#### 問題

デバッガのコントロール下ではプロジェクトは正常に動作するが、デバッガなしでプログラムを実行するとクラッシュする。

#### 背景

デバッガ上での実行は、プログラムの動作環境であるオペレーティング環境を変更してしまいます。正確に動作する、バグのあるプログラムを作成してしまう可能性もあります。何が問題なのかを特定することは困難ですが、以下に奇妙な挙動の原因となりうる事柄を記します。

デバッガは実行速度を低下させます。時間に敏感なコードがある場合、デバッガなしでは実効速度が速すぎるかもしれません。これを頭に入れて問題を追いかけてください。

プロジェクトのメモリ管理もデバッガによって影響されます。デバッガの動作中にメモリブロックを移動することはできません。デバッガが動作していない場合、ブロックは移動可能になり、メモリの再配置によるバグが発生します。

#### 解決方法

レース状況などの時間に敏感なコードに注目してください。

メモリに関する問題に注意してください。ヌルポインタやハンドルへのアクセス、リソースハンドルの不正な処理、複数回のハンドルの処理などです。

ローレベルのデバッグテクニックをみがいてください。

## ブレークポイントの問題

ここではブレークポイントの設定、消去に関する問題について説明します。

### ステートメントにブレークポイントを設定できない

#### 問題

ステートメントのブレークポイント列にバー(-)がなく、ブレークポイントを設定できない。

#### 背景

CodeWarrior リンカは非常によくできています。最終的な製品にリンクされないソースコードのシンボル情報は生成しません。実際に使用されないステートメントは最終的なオブジェクトコードにインクルードされません。このようなステートメントにはオブジェクトコードがないため、ブレークポイントを設定することはできません。

コードの最適化はオブジェクトコードを並べ換えるため、オブジェクトコードとソースコードの対応関係に影響を与えます。このためブレークポイントを正確に設定することは非常に困難になります。

#### 解決方法

コードがすべて使用されているかを確認してください。必要ならばソースコードを変更してください。

コンパイラ疑似命令のせいでコンパイラがステートメントを無視していないか、ソースコードを見て確認してください。

すべてのコンパイラの最適化をオフにしてください。『Instruction Scheduling』をオフ、『Don't Inline』をオンにしてプロジェクトを再度ビルドしてください。最適化によってオブジェクトコードが変更され、ソースコードと対応しなくなることがあります。

一行のソースには一つ一つのステートメントを入れるようにコードを書いてください。コンパイラは一行に複数のステートメントがあってもブレークポイント情報を出力できます。しかしデバッガはソース行を表示するだけなので、同じ行を複数回ステップ実行してしまいます。

[『ブレークポイントでコードを最適化』\(p66\)](#) を参照してください。

## ブレークポイントが反応しない

### 問題

設定したブレークポイントが働かない。

### 背景

ブレークポイントがアクティブである場合、条件（もしあれば）が TRUE である場合は、ブレークポイントに達すると実行を停止します。

### 解決方法

コードをステップ実行して、ブレークポイントを設定したステートメントに達しているかを確認してください。

ブレークポイントウィンドウでブレークポイントがアクティブであるか、確認してください。

ブレークポイントが条件付きである場合、それが TRUE であることを確認してください。デバッガは条件が FALSE のブレークポイントを無視します。

[『ブレークポイントを設定』\(p64\)](#)、[『条件ブレークポイント』\(p66\)](#) を参照してください。

## 変数の問題

ここでは変数に関係のある問題について説明します。

## 変数が変化しない

### 問題

変数に値を割り当てたが、デバッガで変数が変化しない。

### 背景

コンパイラはコード内で使用されない変数を削除します。

### 解決方法

未使用の変数をコードから除去してください。

変数を使用するようにコードを修正してください。

## 変数に誤った値が割り当てられる

### 問題

変数の変化が不正確である。以下のいずれかの問題に遭遇している。

二つ以上の変数に対して、同時に同じ値が設定される。

ある変数が、他の変数に割り当てられるべき値を受け取っている。

### 背景

コンパイラは変数が同時に使用されていないことを識別し、同じロケーションに変数を格納しています。これはコンパイラが自動的に行う最適化で、『レジスタカラーリング』と呼ばれます。レジスタカラーリングは関数内で変数がどのように使われているかを調べます。二つ以上の変数が同じ有効範囲内にあり、かつ同時に使われない場合、コンパイラはそれらの変数に対して同じプロセッサレジスタを使います。複数の変数に、メモリではなくレジスタを使うことによってプログラムのパフォーマンスが改善されます。

[リスト 8.1](#) はレジスタカラーリングを誘発するコードの例です。異なる四つの変数が設定されていますが、同時には使用されません。コンパイラは四つの変数に同じレジスタを使います。四つの変数が同じレジスタにあることはデバグにはわからないため、四つの変数すべてが変化していることを表示します。実際には[リスト 8.1](#) のコードは何もしません。最適化によって削除されるからです。

### リスト 8.1 レジスタカラーリングで変化する変数

---

```
void main(void)
{
    long a = 0, b = 0, c = 0, d = 0;

    a = 1; /* a is set to 1 */
    b = 2; /* a is set to 2, b remains unchanged */
    c = 3; /* a is set to 3, c remains unchanged */
    d = 4; /* a is set to 4, d remains unchanged */
}
```

---

### 解決方法

そのままにしておきます。レジスタカラーリングは問題ではありません。

C/C++ でレジスタカラーリングを防ぐには、`volatile` キーワードで変数を宣言します。これはプリプロセッサ疑似命令で行ってください。そうするとデバグの後に `volatile` 格納クラスの指定子を簡単に除去することができます。

『C/C++ Language Manual』、『Assembly Language Manual』を参照してください。

## 奇妙な変数

### 問題

ローカルおよびグローバル変数欄に、ソースコードで宣言していない変数が表示される。

### 背景

コンパイラはソースコードを翻訳するときに、オブジェクトコードの中に独自のテンポラリ変数を生成することがあります。このような テンポラリ 変数はドルマーク (\$) 付きの名前で表示されます。また C++ の仮想基底クラス型も \$ の接頭子付きで表示されます。

コンパイラとリンカはプログラムの初期化と終了を支援するため、ライブラリおよびランタイム関数から変数を追加することがよくあります。

### 解決方法

ありません。これは解決が必要な問題ではありません。

## 奇妙なデータ型

### 問題

Data メニューで『Show Types』を選択したとき、enum 型の値が『?anonx』型 (x は任意の数) のように表示される。

### 背景

enum 型の名前がソースコードで定義されていなければ、デバッガはその名前を表示できません。コンパイル時に、コンパイラは enum 型に generic 型の名前を割り当てます。これはデバッガが表示可能な一般的な名前です。

例えば [リスト 8.2](#) で、『Show Types』を選択すると、変数 myMarx は enum 型『?anonx』を持つかのように表示されます。これは enum 型に名前がないためです。一方変数 myBeatle は Beatle 型として表示されます。enum 型がその名前で定義されているためです。

### リスト 8.2 名前のない enum 型 (C/C++)

---

```
// Debugger displays as anonymous type
enum {Groucho,
      Harpo,
      Chico,
      Zeppo } myMarx = Harpo;

// Debugger displays as type Beatle
typedef enum Beatle {John,
                    Paul,
                    George,
                    Ringo} myBeatle = John;
```

---

## 解決方法

ありません。これは解決が必要な問題ではありません。

## データ型が認識されない

### 問題

独自のデータ型を宣言したが、その型で変数を見ることができない。

### 背景

シンボルファイルにはプログラムで使用される型の情報だけが含まれます。typedef で定義された型はシンボルファイルには格納されません。このため、派生した型のように変数を表示する必要があります。例えば、long データ型に基づいて MyLong 型を宣言した場合、( MyLong ではなく ) long 型として表示されます。

[『データを異なる型で表示する』\(p75\)](#) を参照してください。

## 解決方法

基本のデータ型を使ってください。

Data メニューの『Show Types』を選択し、デバッガがどのように型を識別しているかを調べてください

## Expression ウィンドウの『未定義の識別子』

### 問題

Expression ウィンドウの評価式にあるユーザー定義型の値が『・ undefined identifier ・』になる。

### 背景

デバッガは、単純に他の型のエイリアスであるデータ型を認識できません。エイリアス型はシンボルファイルに含まれないためです。

パスカル形式の宣言を例にしてみます。

```
TYPE
    MYBIGINT = LONGINT;
```

### 評価式は

```
MYBIGINT(thePtr)
```

デバッガの Expression ウィンドウには値が『・ undefined identifier ・.』として表示されます。結果を修正するためには次の評価式を使ってください。



```
LONGINT(thePtr)
```

#### 解決方法

定義したデータ型ではなく、オリジナルのデータ型を使ってください。

[『評価式の制限』\(p86\)](#) を参照してください。

## ソースファイルの問題

ここではソースファイルに関係のある問題について説明します。

### ソースコードが表示されない

#### 問題

ソース欄にアセンブリ言語のコードしか表示されない。ソースポップアップメニューからもソースコードを表示できない。

#### 背景

そのコードのシンボル情報がありません。ファイルに対してデバッグ機能をオンにしていない、またはリンクが追加したソースコードと対応しない補助的なコード（グルーコードなど）を実行している可能性があります。シンボル情報がなければ、デバッガはアセンブラコードしか表示できません。

#### 解決方法

コードが独自のソースファイルにある場合、CodeWarrior IDE がそのファイルのシンボル情報を生成しているかを確認してください。

他のソース（コンパイルされたライブラリなど）のコードでは、関数からステップアウトして呼び出しもとへ戻ってください。表示するソースコードはありません。

[『デバッグのためにファイルを設定』\(p16\)](#) を参照してください。

### ソースファイルの修正日時

#### 問題

プロジェクトを実行すると、『修正日時が一致しない』という警告メッセージが表示される。

#### 背景

シンボルファイルはソースファイルが最後にいつ変更されたかを追跡しています。シンボルファイルに保存された修正日時がオリジナルのファイルのものと一致しなければ、デバッガはシンボル情報が古いという警告を発します。

#### 解決方法

ソースファイルをタッチ（または変更を加えずにメイクして保存）してからプロジェクトを再度ビルドするか、ファイルをアップデートしてください。

### プロジェクトでソースコードを共有

#### 問題

二つの異なるブラウザウィンドウに同じソースファイルを表示しようとするすると警告が表示される。

#### 背景

異なるブラウザウィンドウに同じソースファイルを表示することはできません。

#### 解決方法

ファイルのコピーを作り、各プロジェクトに追加してください

### Pascal プロジェクトの ANSI C コード

#### 問題

Pascal で書いたコードをステップ実行すると ANSI C 関数が見つかった。ANSI C ライブラリもインクルードしていない。

#### 背景

Pascal のランタイムライブラリは、ANSI C 関数を使って C で記述されています。Pascal プログラムをデバッグするときにこれらの関数が見つかります。

#### 解決方法

ありません。これは解決が必要な問題ではありません。

## デバッガのエラーメッセージ

以下にデバッガのエラーメッセージの一覧およびエラーの発生原因のヒントを示します。

An unknown error occurred while trying to target an existing process.

既存のプロセスをターゲットにしようとしたときに未知のエラーが起きました。

Bad type code

内部エラーです。

## Bus Error

不当なアドレスを読み書きしようとしています。

can't display value -- type information not supported

シンボルファイルに CodeWarrior デバッガがサポートしていないデータ型が含まれています。

Can't use this source file, it was not saved before running, or was edited after linking.

コンパイラが扱っている最中のファイルをデバッガがアクセスすることはできません。デバッグ情報が存在しないテキストを参照しているのでなければ、デバッガは警告を発するだけです。参照していれば、このエラーメッセージが表示されます。

class name expected

評価式を評価中にクラス名が想定される部分に、それ以外のものが見つかりました。

Could not complete your request because the process is not suspended.

プログラムを実行しているときに、該当コマンドを実行することはできません。

Could not set a watch point because the page containing that memory location overlaps low memory or the system heap.

Watchpoint は、ローメモリやシステムヒープに設定できません。

Could not set a watch point because the page containing that memory location overlaps the stack.

Watchpoint はメモリのライトプロテクショナルアルゴリズム（ページレベルの操作）に基づいてインプリメントされています。スタックを含むページメモリへの書き込みをプロテクトすることはできません。

Couldn't locate the program entry point, program will not stop on launch.

プログラムを起動するとき、デバッガは暗黙のブレークポイントを `main()` (C/C++) 関数または `main` プログラム (Pascal) の前に設定します。そのような部分がないときは、停止せずにただ実行するだけです。

identified or qualified name expected

評価式を評価中に、識別子または定義済みの名前以外のものが見つかりました。

illegal character constant

評価式を評価中に、不当な文字定数が見つかりました。

illegal string constant

評価式を評価中に、不当な文字列定数が見つかりました。

illegal token

評価式を評価中に、不当なトークンが見つかりました。

Invalid C or Pascal string.

評価式を評価中に、評価できない文字列が見つかりました。

Invalid character constant.

評価式を評価中に、評価できない文字定数が見つかりました。

Invalid escape sequence inside string or character constant.

文字列中の C/C++ のエスケープシーケンスは無効なシンタックスです。

invalid pointer or reference expression

評価式を評価中に、評価できないポインタまたは参照式が見つかりました。

invalid type declaration

評価式を評価中に、評価できない型が見つかりました。

invalid type information in SYM file

シンボルファイルに含まれるデータが不当なので、CodeWarrior デバッガは変数を表示できません。

New variable value is too large for the destination variable.

例えば、10 バイトの文字列変数に 20 バイトの文字列を割り当てようとしてしました。

No type with that name exists.

CodeWarrior デバッガは View As ダイアログで指定した型を認識しません。

Register not available

デバッガはレジスタ変数を表示するために有効なレジスタを取得できません。例えば、カレントルーチンからスタック上にあるルーチンを探すとき、(スタックの下にあるルーチンがデバッグ情報を持たない限り) デバッガは保存しているレジスタの値を見ることができません。

string too long

文字列長が長すぎます。

The new variable value is the wrong type for the destination variable.

変数に間違った型を割り当てようとしています。

typedef name expected

評価式を評価中に、`typedef` 名を想定しているのに別のものが見つかりました。

Unable to step from here.

デバッガはこのポイントからステップ実行できません。

Unable to step out from here.

デバッガはこのポイントからステップアウトできません。

undefined identifier

評価式を評価中に、定義されていない識別子が見つかりました。

unexpected token

評価式を評価中に、予期しないトークンが見つかりました。

unknown error "^0"

内部エラーです。

unterminated comment

コメントが終了していません。

Variable or expression cannot be used as an address.

例えば、`r` が `short` 型るとき、`*(char*)r` は無効です。

Warning - this SYM file has some invalid or inconsistent data. The debugger may show incorrect information.

シンボルファイルが壊れている可能性があります。

'\*' or '&' expected

評価式を評価中に、予期しないポインタまたは参照オペレータが見つかりました。



# 索引

---

## 記号

\$ (変数名の) 127

? (変数名の) 127

## A

### ANSI

Pascal プロジェクトの C コード 130

エスケープシーケンス 116

Apple メニュー 119

## B

Break on C++ exception コマンド 112

Breakpoints Window コマンド 65

## C

C String コマンド 116

### C++

オブジェクトコンストラクタを無視 101

デバッグ 96, 99

メソッドをアルファベット順に並べる 95

メソッドをソート 32

Character コマンド 116

Clear All Breakpoints コマンド 64, 112

Clear Current Watchpoint コマンド 70, 114

Clear Watchpoint コマンド 37, 114

Clear コマンド 35, 37, 70, 110

Close All Variable Windows コマンド 33, 118

Close コマンド 108

Collapse All コマンド 72, 113

Control メニュー 110

Copy to Expression コマンド 35, 78, 113

Copy コマンド 37, 109

Cut コマンド 109

### C 文字列

としてデータを入力 77

としてデータを表示 75

### C 言語

エスケープシーケンスの入力 116

文字列を表示 116

## D

Data メニュー 112

Debugger Settings 設定パネル 104

Debugger 環境設定ファイル 112

Debug コマンド 18, 122

Default コマンド 116

Disable Debugger コマンド 15

## E

Edit コマンド 61, 108

Edit メニュー 109

Enable Debugger コマンド 15, 122

Enumeration コマンド 117

Expand コマンド 72, 113

Expressions Window コマンド 35, 78

Expression ウィンドウ

項目を追加 78

順序の変更 78

定義 35

と変数 78

呼び出し元の変数を追加 78

## F

File メニュー 107

Find Next コマンド 61, 62, 63, 110

Find Selection コマンド 62, 63, 110

Find コマンド 61, 62, 110

Fixed コマンド 117

Floating Point コマンド 116

FPU Registers コマンド 41

FPU レジスタ 80

FPU レジスタウィンドウ 28

Fract コマンド 117

## G

General Registers コマンド 41

## H

Hexadecimal コマンド 116

Hide Breakpoints コマンド 118

Hide Expressions コマンド 118

Hide FPU Registers コマンド 118

Hide Processes コマンド 117

Hide Registers コマンド 118  
Hide Watchpoints コマンド 118

## K

Kill コマンド 24, 50, 52, 56, 111

## L

Log System Messages オプション 104  
Log ウィンドウ 37  
lvalue 40

## M

Macintosh  
    ROM モニタプログラム 112  
MacBug  
    CodeWarrior デバッガとの切り替え 112

## N

New Expression コマンド 113

## O

ObjectSupportLib 12  
Open Array Window コマンド 33, 113  
Open Variable Window コマンド 33, 113  
Open コマンド 107

## P

Pascal String コマンド 116  
Pascal,C コード 130  
Pascal 文字列  
    としてデータを表示 75  
    としてデータを入力 77  
Paste コマンド 109  
PPCTraceEnabler 13  
Preferences コマンド 110  
Processes Window コマンド 43  
Program Arguments オプション 104  
Program to Launch for Debugging Shared Libs, DLLs  
    and Code Resources オプション 104

## Q

QC-aware (Mac OS) 101  
Quit コマンド 109

## R

RAM Doubler 69  
ResEdit 112  
Run コマンド 18, 24, 50, 51, 108, 110, 122

## S

Save A Copy As コマンド 108  
Save As コマンド 37, 108  
Save コマンド 37, 108  
Select All コマンド 110  
Set Watchpoint コマンド 69, 114  
Show Breakpoints コマンド 35, 118  
Show Expressions コマンド 118  
Show FPU Registers コマンド 41, 80, 118  
Show Processes コマンド 117  
Show Registers コマンド 80, 118  
Show Types コマンド 113, 127  
Show Watchpoints コマンド 36, 118  
Show/Hide Toolbar コマンド 25  
Signed Decimal コマンド 116  
Solaris 10  
    キーボードの表記 10  
Speed Doubler 69  
static 101  
Step Into コマンド 24, 50, 53, 111  
Step Out コマンド 24, 50, 53, 111, 112  
Step Over コマンド 24, 50, 52, 111  
Stop コマンド 24, 50, 51, 55, 111  
Switch to Monitor コマンド 112  
Switch To MW Debugger コマンド 61

## T

\_\_throw() 112

## U

Undo コマンド 109  
Unsigned Decimal コマンド 116

## V

View As コマンド 114  
View Memory As コマンド 38, 40, 79, 115  
View Memory コマンド 39, 79, 115



## W

Watchpoints Window コマンド 70

Watchpoints ウィンドウ 36

開く 36

Working Directory オプション 104

## ア行

アクティブな欄 22

アセンブラ

表示 27, 34

メモリを表示 28

レジスタを表示 28

一行ずつ実行する 52

一時的なブレークポイント 54

設定 65

普通のブレークポイントへの影響 65

一般的なコードナビゲーション 57

ウォッチポイント

68K システムの 69

Watchpoints ウィンドウ 36

を表示 70

消去 37, 70

制限 69

設定 69

定義 68

エスケープシーケンス

で文字を表示 116

入力 116

エラー

QC 101

オプション

QC-aware 101

## カ行

拡張

変数を 23, 39, 72, 113

仮想メモリ

とデバッグ 123

カレントステートメント矢印 25, 111

定義 51

ドラッグ 54

ブラウザウィンドウ 33

ブレークポイントの 63

環境設定

Settings & Display 93

環境設定パネル

Always prompt for source file location if file not found オプション 98

At startup, prompt for SYM file if none specified オプション 98

Attempt to use dynamic type of C++ or Object Pascal objects オプション 96

Automatically launch applications when SYM file opened オプション 99

Confirm "Kill Process" when closing or quitting オプション 99

Default size for unbound arrays オプション 97

Display 95

Don't step into runtime support code オプション 101

Ignore file modification dates オプション 99

In variable panes, show variable types by default オプション 95

Save breakpoints オプション 94

Save expressions オプション 95

Save window settings in local ".dbg" files オプション 94

Show variable values in source code オプション 95

Sort functions by method name in browser オプション 95

Stop at program main when launching applications オプション 100

Use temporary memory for SYM data オプション 98

関数ポップアップメニュー 30, 34

アルファベット順 34

アルファベット順に表示 29

関数欄 30, 34

画面図について 9

キーボードについて 9

キーボードの表記

Solaris 10

起動

デバッグを 47

起動、アプリケーションを

自動的に 99

局所変数

デバッグで表示 72

局所変数欄 33

切り替える、プロジェクト環境へ 61

クリップボード

デバッグでの使用 110

無限ループ

から抜ける 55  
コールチェーンによるナビゲーション 57  
混合表示 28

## サ行

サイズ変更  
    欄を 22, 30  
削除  
    評価式を 35  
修正日時  
    デバッグの 99  
終了、実行を 56  
    停止との違い 56  
消去  
    ブレークポイントを 64, 36  
シンタックスカラー 63  
シンボルファイル  
    定義 16, 19  
    デバッグ 15  
    内容 19  
    開く 19  
    複数のファイルを開く 30  
    開く 107  
実行を停止する 55  
順序を変える  
    評価式の 35  
条件付きブレークポイント 36, 66  
    評価式と 84  
    ループと 66  
条件ブレークポイントを設定 66  
スキップ、ステートメントを 54  
スタック  
    関数コールの表示 22  
スタッククロール欄 22, 57  
static コンストラクタ  
    をデバッグ 99, 101  
ステップ実行  
    ランタイムコード内を 101  
    コード 52  
    ルーチンの外へ 53  
    ルーチンの中へ 53  
条件ブレークポイント  
    設定 66  
設定  
    ブレークポイントを 36, 64  
設定パネル

Show Variable Types by Default オプション 74  
Use temporary memory for SYM data オプション 123

## 選択

    欄内の項目を 30  
選択、欄での 22  
ソースコード  
    ナビゲーション 60  
    フォントと色 63  
ソースファイルの位置 98  
ソースポップアップメニュー 30, 34  
ソース欄 25, 33

## タ行

ターゲット  
    デバッグを準備 15  
大域変数  
    変数欄の 72  
    ローカル欄の 23  
大域変数欄 30, 72  
タスク欄 45  
ダンプ  
    メモリを 115  
ツールバー  
    デバッグ 24  
停止、実行を  
    終了との違い 56  
テンポラリ変数 127  
テンポラリメモリ  
    デバッグの使う 123  
データ型  
    anon 127  
    enum 型 127  
    キャスト 114  
    構造体を表示 23  
    表示 74, 113  
    複数の 78  
    利用可能な 75  
データフォーマット  
    変数の 77  
デバッグ  
    起動時の問題 122  
    起動 47  
    ツールバー 24  
    定義 7, 47  
    フォントの選択 63

- プロジェクトから起動 18
- ローレベル 112
- デバッガ, 統合 10
- デバッガを起動
  - プロジェクトから 18
- デバッグ
  - C++ を 96
  - static コンストラクタを 99
  - ターゲットを準備 15
  - ファイルを準備 16
- デバッグの準備 15
- デバッグ列、プロジェクトウィンドウ 17
- 統合デバッガ 10
- トラブルシューティング
  - Run コマンド 122
  - 起動が遅い 123
  - 起動時のエラー 123
  - 奇妙なデータ型 127
  - 奇妙な変数名 127
  - ソースコードがない 129
  - ソースファイルが古い 129
  - データ型 128
  - デバッガの起動 122
  - バスエラー 125
  - ブレークポイント 124, 125
  - 変数が変化しない 125
  - 変数値の変更 126
  - 未定義の識別子 128
- トレース 53
- ナ行
- ナビゲーション
  - 一般的な 57
  - コールチェーンによる 57
  - ブラウザウィンドウによる 58
- ナビゲート
  - ソースコードによる 60
  - ファイル欄で 59
- 入力、データを
  - フォーマット 77
- 八行
- 配列
  - サイズを設定 97
- 配列ウィンドウ 38, 73
  - 先頭アドレスを設定 38
- 評価式
  - Expressions ウィンドウの 83
  - 公式のシンタックス 88
  - 削除 35
  - 作成 83
  - 制限 86
  - ソースアドレスとしての 85
  - 定義 83
  - 定数 87
  - 特別な機能 85
  - と構造体メンバ 88
  - と変数 87
  - ドラッグ 84
  - 内のポインタ 88
  - ブレークポイントウィンドウの 84
  - ブレークポイントに付ける 84
  - メモリウィンドウの 85
  - 例 88
  - レジスタと 86
  - 論理式 88
- 評価式の順序を変える 35
- 表記規則
  - マニュアル 8
- 表示
  - アセンブラコードを 27, 34
  - アセンブラとソースを 28
  - アドレスを指定してメモリを 79
  - ウォッチポイントを 70
  - 局所変数を 72
  - コールチェーンを 22
  - 大域変数を 33, 72
  - データを複数の型で 78
  - デフォルトによる変数型の 74
  - ブレークポイントを 65
  - ポインタ型として 76
  - メモリを 74
  - レジスタ 42
  - レジスタを 28, 41
- 開く
  - シンボルファイルを 107
- ファイル
  - デバッグを準備 16
- ファイル修正日時を無視 99
- ファイル欄 30, 31, 33
  - でコードのナビゲート 59
  - と大域変数 31, 72
- フォーマット
  - データ入力の 77
- フォントの選択、デバッガの 63

- 複数のデータ型 78
- ブラウザのソース欄 30
- ブラウザウィンドウ
  - コードのナビゲート 58
  - ブレークポイントの設定 33
  - プログラムウィンドウとの比較 29
- ブレークポイント
  - 一時的な 54
  - 一時的なブレークポイントの影響 65
  - 消去 36, 64
  - 条件付き 36, 66
  - 条件付き ~ とループ 66
  - 条件付きブレークポイント 84
  - すべてを消去 112
  - 設定 64, 125
  - 設定できない 124
  - 定義 63
  - 表示 65
  - ブラウザウィンドウでの設定 33
  - ブレークポイントウィンドウでの設定 36
  - 一時的な 65
  - 条件, 設定 66
- ブレークポイントウィンドウ 35
- プログラムウィンドウ
  - 起動時の 47
  - 内容 21
  - ブラウザウィンドウとの比較 29
- プロジェクトウィンドウのデバッグ列 17
- プロジェクトでファイルを共有 130
- プロセス欄 44
- プロセッサレジスタ 80
- 変更
  - デバッガのフォントと色を 63
  - 変数値を 76
  - メモリ変更の危険性 41
  - メモリを 40
  - レジスタを 41
- 変数
  - enum 型 117
  - Expression ウィンドウの 78
  - ウィンドウを開く 33
  - 拡張する 113
  - 奇妙な名前 127
  - 局所 23, 72
  - 自動的にウィンドウを閉じる 37
  - 静的 33
  - 大域 31, 72
  - 値を変更 76

- テンポラリ 127
- データフォーマット 77
- 別ウィンドウに表示 33
- 別のウィンドウに表示 73
- 変数ウィンドウ 37, 73
- 変数欄 23, 72
  - と大域変数 72
- 変数を拡張 23, 39, 72
- ホストの表記 9
- ポインタ型 76

## マ行

- マニュアルの表記規則 8
- 未定義の識別子 128
- 無限ループ
  - 回避 56
  - から抜ける 56
- メソッド (C++)
  - アルファベット順に並べる 95
  - ソート 32
- メモリウィンドウ 39, 74
  - アドレスを変更 40
  - 内容を変更 40
- メモリダンプ 74, 85, 115
- メモリを変更 40
- 文字列
  - C 文字列として表示 116
  - Pascal 文字列として表示 116
- 戻る、プロジェクト環境へ 61

## ラ行

- 欄 32
  - アクティブな 22
  - 選択 22
- 欄内の項目を選択 30
- 欄をサイズ変更 22, 30
- ルーチン
  - ローカルな 78
- ループ (無限)
  - から抜ける 56
- ループ、無限
  - 回避 56
  - から抜ける 55
- ループと条件付きブレークポイント 66
- レジスタ 80

---

FPU 80  
値を変更 41  
で指されたメモリを表示 79  
評価式内の 86  
表示 28, 41  
値を変更 41, 42  
表示 41, 42  
レジスタウィンドウ 28  
レジスタカラーリング 126  
論理評価式 88



# CodeWarrior Debugger User Guide

## Credits

writing lead: Derek Saldana

other writers: L. Frank Turovich, Gene Backlin,  
Carl Constantine, Marc Paquette, Jim Trudeau,  
Alisa Dean, Roopa Malavally

engineering: Mike Lockwood, Eric Cloninger,  
Honggang Zhang, Joel Sumner

frontline warriors: Howard Katz, John McEnerney,  
Raja Krishnaswamy, Lisa Lee,  
Ron Levreault, Jorge Lopez, Fred Peterson,  
Khurram Qureshi, Eric Scouten, Dean Wagstaff,  
Terry Constantine, Stephen Espy,  
CodeWarrior users everywhere

translation: Naomi Owashi

# CodeWarrior 文書のガイド

CodeWarrior の文書はツールと同様にモジュールのように構成されています。ツール、言語、ライブラリ、ターゲットごとにマニュアルがあります。各 CodeWarrior 製品によって含まれるマニュアルが異なります。この表に記載されていないマニュアルが含まれることもあります。

コアマニュアル	
IDE User Guide	CodeWarrior IDE の使い方
Debugger User Guide	CodeWarrior デバッガの使い方
CodeWarrior Core Tutorials	IDE の各ツールのチュートリアル
言語 / コンパイラのマニュアル	
C Compilers Reference	C/C++ フロントエンドコンパイラの情報
Pascal Compilers Reference	Pascal フロントエンドコンパイラの情報
Error Reference	コンパイラ / リンカのエラーメッセージのリストおよび解説
Pascal Language Reference	Metrowerks における ANS Pascal の実装
Assembler Guide	スタンドアローンアセンブラのシンタックス
Command-Line Tools Reference	Mac OS / BeOS コンパイラのコマンドラインのオプション
Plugin API Manual	CodeWarrior のプラグインコンパイラ / リンカの API
ライブラリのマニュアル	
MSL C Reference	Metrowerks ANSI 標準 C ライブラリの関数のリファレンス
MSL C++ Reference	Metrowerks ANSI 標準 C++ ライブラリの関数のリファレンス
Pascal Library Reference	Metrowerks ANS Pascal ライブラリの関数のリファレンス
MFC Reference	Win32 用の Microsoft Foundation Classes リファレンス
Win32 SDK Reference	Win32 API の Microsoft リファレンス
The PowerPlant Book	Mac OS 用アプリケーションフレームワークのガイド
PowerPlant Advanced Topics	PowerPlant での Mac OS プログラミングの高度なテクニック
ターゲットマニュアル	
Targeting BeOS	BeOS プログラミングでの CodeWarrior の使い方
Targeting Java VM	Java 仮想マシンプログラミングでの CodeWarrior の使い方
Targeting Mac OS	Mac OS プログラミングでの CodeWarrior の使い方
Targeting MIPS	MIPS 組み込みプロセッサプログラミングでの CodeWarrior の使い方
Targeting NEC V800 series	NEC V810/830 プロセッサプログラミングでの CodeWarrior の使い方
Targeting Net Yaroze	Net Yaroze プログラミングでの CodeWarrior の使い方
Targeting Palm OS	Palm OS プログラミングでの CodeWarrior の使い方
Targeting PlayStation OS	PlayStation プログラミングでの CodeWarrior の使い方
Targeting PowerPC	PPC 組み込みプロセッサプログラミングでの CodeWarrior の使い方
Targeting Win32	Windows プログラミングでの CodeWarrior の使い方

印の付いているマニュアルは日本語訳が用意されています。