TEXAS INSTRUMENTS
www.tij.co.jp

参 考 資 料

Code Composer Studio IDE 開発ツール v CCS3.1

入門マニュアル

2005 *Digital Signal Processing Solutions*

JAJU071

Code Composer Studio IDE
入門マニュアル

対応英文マニュアル：SPRU509F
2005年 5月

2005年 6 月

TEXAS INSTRUMENTS

ご注意

日本テキサス・インスツルメンツ株式会社（以下TIJといいます）及びTexas Instruments Incorporated（TIJの親会社、以下TIJおよびTexas Instruments Incorporatedを総称してTIといいます）は、その製品及びサービスを任意に修正し、改善、改良、その他の変更をし、もしくは製品の製造中止またはサービスの提供を中止する権利を留保します。従いまして、お客様は、発注される前に、関連する最新の情報を取得して頂き、その情報が現在有効かつ完全なものであるかどうかご確認下さい。全ての製品は、お客様とTIとの間に取引契約が締結されている場合は、当該契約条件に基づき、また当該取引契約が締結されていない場合は、ご注文の受諾の際に提示されるTIの標準契約約款に従って販売されます。

TIは、そのハードウェア製品が、TIの標準保証条件に従い販売時の仕様に対応した性能を有していること、またはお客様とTIとの間で合意された保証条件に従い合意された仕様に対応した性能を有していることを保証します。検査およびその他の品質管理技法は、TIが当該保証を支援するのに必要とみなす範囲で行なわれております。各デバイスの全てのパラメーターに関する固有の検査は、政府がそれ等の実行を義務づけている場合を除き、必ずしも行なわれておりません。

TIは、製品のアプリケーションに関する支援もしくはお客様の製品の設計について責任を負うことはありません。TI製部品を使用しているお客様の製品及びそのアプリケーションについての責任はお客様にあります。TI製部品を使用したお客様の製品及びアプリケーションについて想定されうる危険を最小のものとするため、適切な設計上および操作上の安全対策は、必ずお客様にてお取り下さい。

TIは、TIの製品もしくはサービスが使用されている組み合せ、機械装置、もしくは方法に関連しているTIの特許権、著作権、回路配置利用権、その他のTIの知的財産権に基づいて何らかのライセンスを許諾するということは明示的にも黙示的にも保証も表明もしておりません。TIが第三者の製品もしくはサービスについて情報を提供することは、TIが当該製品もしくはサービスを使用することについてライセンスを与えるとか、保証もしくは是認するということを意味しません。そのような情報を使用するには第三者の特許その他の知的財産権に基づき当該第三者からライセンスを得なければならない場合もあり、またTIの特許その他の知的財産権に基づきTIからライセンスを得て頂かなければならない場合もあります。

TIのデータ・ブックもしくはデータ・シートの中にある情報を複製することは、その情報に一切の変更を加えること無く、且つその情報と結び付られた全ての保証、条件、制限及び通知と共に複製がなされる限りにおいて許されるものとします。当該情報に変更を加えて複製することは不公正で誤認を生じさせる行為です。TIは、そのような変更された情報や複製については何の義務も責任も負いません。

TIの製品もしくはサービスについてTIにより示された数値、特性、条件その他のパラメーターと異なる、あるいは、それを超えてなされた説明で当該TI製品もしくはサービスを再販売することは、当該TI製品もしくはサービスに対する全ての明示的保証、及び何らかの黙示的保証を無効にし、且つ不公正で誤認を生じさせる行為です。TIは、そのような説明については何の義務も責任もありません。

なお、日本テキサス・インスツルメンツ株式会社半導体集積回路製品販売用標準契約約款もご覧下さい。
http://www.tij.co.jp/jsc/docs/stdterms.htm

弊社半導体製品の取り扱い・保管について

半導体製品は、取り扱い、保管・輸送環境、基板実装条件によっては、お客様での実装前後に破壊/劣化、または故障を起こすことがあります。
弊社半導体製品のお取り扱い、ご使用にあたっては下記の点を遵守して下さい。

1. **静電気**
   - **素手で半導体製品単体を触らないこと。どうしても触る必要がある場合は、リストストラップ等で人体からアースをとり、導電性手袋等をして取り扱うこと。**
   - **弊社出荷梱包単位（外装から取り出された内装及び個装）又は製品単品で取り扱いを行う場合は、接地された導電性のテーブル上で（導電性マットにアースをとったもの等）、アースをした作業者が行うこと。また、コンテナ等も、導電性のものを使うこと。**
   - **マウンタやはんだ付け設備等、半導体の実装に関わる全ての装置類は、静電気の帯電を防止する措置を施すこと。**
   - **前記のリストストラップ・導電性手袋・テーブル表面及び実装装置類の接地等の静電気帯電防止措置は、常に管理されその機能が確認されていること。**
2. **温・湿度環境**
   - **温度：0～40℃、相対湿度：40～85%で保管・輸送及び取り扱いを行うこと。（但し、結露しないこと。）**
   - **直射日光があたる状態で保管・輸送しないこと。**
3. **防湿梱包**
   - **防湿梱包品は、開封後は個別推奨保管環境及び期間に従い基板実装すること。**
4. **機械的衝撃**
   - **梱包品（外装、内装、個装）及び製品単品を落下させたり、衝撃を与えないこと。**
5. **熱衝撃**
   - **はんだ付け時は、最低限260℃以上の高温状態に、10秒以上さらさないこと。（個別推奨条件がある時はそれに従うこと。）**
6. **汚染**
   - **はんだ付け性を損なう、又はアルミ配線腐食の原因となるような汚染物質（硫黄、塩素等ハロゲン）のある環境で保管・輸送しないこと。**
   - **はんだ付け後は十分にフラックスの洗浄を行うこと。（不純物含有率が一定以下に保証された無洗浄タイプのフラックスは除く。）**

以上

2001.11

目次

表

最初にお読みください

本書について

Code Composer Studio IDE™ を使用する前に、本書の最初の 2 つの章をお読みください。他の章には、特定のプロセッサおよびツールに関する詳細な情報が含まれています。これらの機能を利用できるかどうかについては、Code Composer Studio をインストールすると提供されるオンライン・ヘルプを参照してください。

第 1 章

概要

この章では、TI eXpressDSP テクノロジー・イニシアチブについて説明します。また、Code Composer Studio IDE を用いた基本的な開発フローを示します。

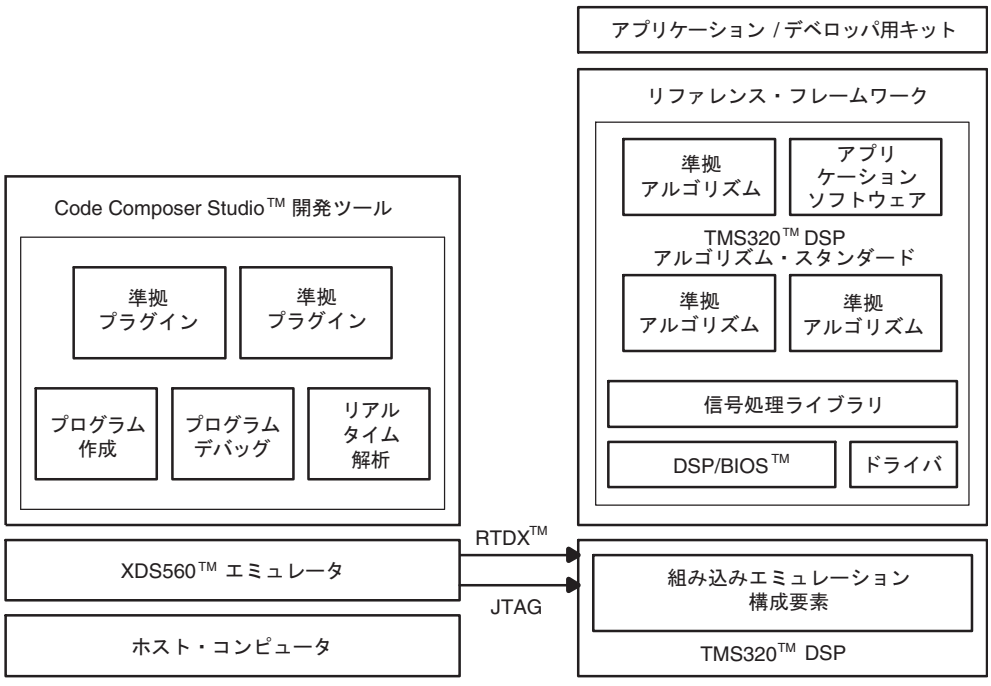
項目 **ページ**

1.1 eXpressDSP™ の紹介

TI では、さまざまな開発ツールを提供しています。これらのツールを使用すると、ディジタル・シグナル・プロセッサ（DSP）ベースのアプリケーション設計プロセスである、コンセプトからコード作成 / ビルド、デバッグ機能を使用した解析、チューニング、およびテストまでをすばやく行うことができます。これらのツールの多くは、TI のリアルタイム eXpressDSP™ ソフトウェア＆開発ツール戦略の一環をなすものです。これらのツールを使用することにより、設計プロセスにおける貴重な時間を節約するばかりでなく、すばやい商品化が可能になります。TI のリアルタイム eXpressDSP™ ソフトウェア＆開発ツール戦略は、開発者が TMS320™ DSP の可能性を最大限に引き出すことができるようにする、密接に関連した 3 つの要素から構成されています。

- 強力な DSP 統合開発ツール：Code Composer Studio IDE
- eXpressDSP は、次の要素から構成されています。
  - スケーラブルな、リアルタイム・ソフトウェア基盤：DSP/BIOS™ カーネル
  - アプリケーションの相互運用性と再利用のための規格：TMS320 DSP アルゴリズム・スタンダード
  - DSP 設計をすばやく開始できる、多数のアプリケーションに共通したソース・コード：eXpressDSP リファレンス・フレームワーク
- TI の DSP サードパーティ・ネットワークから、容易にシステムに統合できる eXpressDSP 準拠製品を含む、TI の DSP をベースとする製品が増加し続けています。

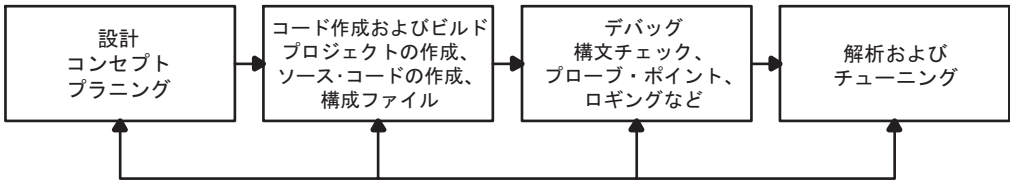
図 1-1. eXpress DSP™ ソフトウェア & 開発ツール



1.2 開発フロー

DSP ベースのアプリケーションの開発フローは、4 段階の基本的なフェーズ、アプリケーション設計、コード作成、デバッグ、および解析 / チューニングから構成されています。この入門マニュアルでは、Code Composer Studio を使用する場合のプログラム開発フローにおける基本的な手順および手法について説明します。

図 1-2. 簡略化した Code Composer Studio IDE 開発フロー



第 2 章

クイック・スタート

この章では、Code Composer Studio の基本的な特徴と機能性の概要を示します。この章をお読みいただくと、簡単なプロジェクトを作成し、ビルドする手順がわかるようになります。経験のあるユーザーはこの章をスキップし、Code Composer Studio のさまざまな特徴を詳細に説明している第 3 章以降へ進んでください。

項目 **ページ**

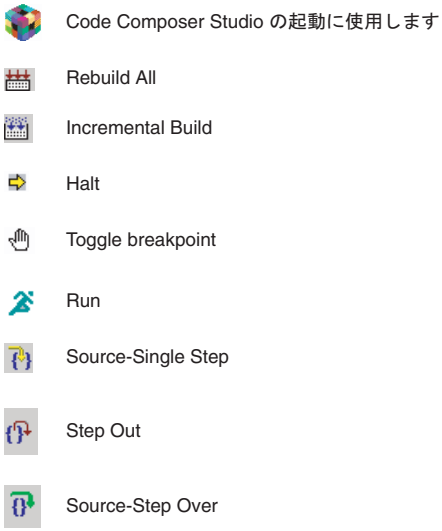
2.1 Code Composer Studio IDE の起動

Code Composer Studio IDE を初めて起動する場合は、デスクトップ上にあるアイコン（下記参照）をダブルクリックします。シミュレータは、デフォルトで自動的に構成されます。Code Composer Studio を特定のターゲットに対して構成する場合の詳細は第 3 章を参照してください。

2.1.1 Code Composer Studio で使用されるアイコン

本書全体で、これらのアイコンを使用しています。

図 2-1. IDE ツールバー上のアイコン



2.2 新規プロジェクトの作成

作業プロジェクトを作成する手順は、次のとおりです。

1. Code Composer Studio を C:¥CCStudio_v3.1 にインストールした場合、C:¥CCStudio_v3.1¥myprojects フォルダ内に practice という名前のフォルダを作成します。
2. この新しいフォルダに C:¥CCStudio_v3.1¥tutorial¥target¥consultant フォルダ内のコンテンツをコピーします。ターゲットは、Code Composer Studio の現在の構成内容を参照します。デフォルト構成はないので、Code Composer Studio をスタートする前に、構成をセットする必要があります。Code Composer Studio の構成に関する詳細は、第 3 章を参照してください。
3. 「Project」メニューから、「New」を選択します。
4. 「Project Name」フィールドに、プロジェクト名 practice を入力します。
5. 「Location」フィールドに、ステップ 1 で作成したフォルダ名を入力するか、参照します。
6. デフォルトでは、Project Type は Executable (.out) に設定され、Target は Code Composer Studio の現在の構成内容に設定されます。
7. 「Finish」をクリックします。Code Composer Studio は、practice.pjt という名前のプロジェクト・ファイルを作成します。このファイルは、プロジェクトの設定内容とプロジェクトで使用されるさまざまなファイルの参照を保存します。
8. 「Project」メニューから「Add Files to Project」を選択して、ファイルをプロジェクトに追加します。また、左側の「Project View」ウィンドウ内にあるプロジェクトを右クリックして、「Add Files to Project」を選択することもできます。

9. 作成したフォルダから、main.c、DoLoop.c、および lnk.cmd（セクションをメモリにマップするリンカ・コマンド・ファイル）を追加します。C:¥CCStudio_v3.1¥c6000¥cgtools¥lib¥ ディレクトリを参照し、構成するターゲット用の rts.lib ファイルを追加します。
10. インクルード・ファイルは、ビルド・プロセス中に依存関係をスキャンする際に、プログラムが自動的に特定するため、インクルード・ファイルをプロジェクトに手作業で追加する必要はありません。プロジェクトをビルドすると、「Project View」ウィンドウにインクルード・ファイルが表示されます。

2.3 プログラムのビルド

このように機能するプログラムを作成できましたので、作成されたプログラムをビルドできます。初めてプロジェクトをビルドすることになるので、Build All 機能を使用します。「Output」ウィンドウには、ビルド・プロセスおよびステータスが表示されます。ビルドが完了すると、「Output」ウィンドウに「Build complete 0 errors, 0 warnings, 0 Remarks.」と表示されます。

主に Rebuild All コマンドを使用してプロジェクトを再ビルドするのは、プロジェクト・オプションまたはプロジェクト内の一部のファイルを変更した場合です。詳細は、2.3 節を参照してください。

2.4 プログラムのロード

プログラムが正常にビルドされたら、「File」→「Load Program」の順に選択してそのプログラムをロードします。デフォルトでは、Code Composer Studio IDE は、プロジェクト・ディレクトリ内に Debug という名前のサブディレクトリを作成し、その中に .out ファイルを保存します。practice.out ファイルを選択し、「Open」をクリックしてプログラムをロードします。

> **注：**
> **変更を加えた際に、プロジェクトを再ビルドする場合は、「File」→「Reload Program」の順に選択して、プログラムをリロードすることを忘れないでください。**

2.5 基本的なデバッグ

ここでは Code Composer Studio のデバッガのさまざまな操作方法について説明します。詳細は、第 5 章を参照してください。

2.5.1 Main の表示

Main 関数の実行を開始するには、「Debug」→「Go Main」の順に選択します。プログラムの実行が、Main 関数で停止し、関数の横にある左側の余白内にプログラム・カウンタ（黄色の矢印）が表示されます。この余白部分は、選択マージンといいます。

2.5.2 ブレークポイントの使用

ブレークポイントを設定するには、カーソルを目的の行に移動させ、F9 を押します。また、「Toggle Breakpoint」ツールバー・ボタンを選択してブレークポイントを設定することもできます。ブレークポイントを設定すると、赤いアイコンが選択マージン内に表示されます。ブレークポイントを解除するには、再度 F9 を押すか、「Toggle Breakpoint」ツールバー・ボタンを選択します。

main.c では、「DoLoop(Input1, Input2, Weights, Output, LOOPCOUNT);」の行でブレークポイントを設定します。プログラムの実行が、Main 関数で停止しているため、プログラムを実行するには F5 を押すか、「Debug」→「Run」の順に選択するか、「Run」ツールバー・ボタンを選択します。プログラムの実行は、ブレークポイントに到達すると、停止します。

2.5.3 ソースのステップ実行

ソースのステップ実行は、プログラムの実行が停止している場合のみ可能です。ブレークポイントで停止しているため、ソースのステップ実行を使用するとプログラムを一行毎に実行できます。「Source-Single Step」ボタンを選択して、DoLoop 関数にステップインします。何度かステップ実行して、プログラムの動きを確認します。「Single Step」ボタンの下にあるステップオーバー（Step Over）およびステップアウト（Step Out）機能も使用可能です。アセンブリのステップ実行も可能です。ソースのステップ実行はコード行を一行ずつ実行しますが、アセンブリのステップ実行はアセンブリ命令を一行ずつ実行します。アセンブリのステップ実行に関する詳細は、5.2.1 項を参照してください。

2.5.4 変数の表示

デバッグ・プロセスでは、関数が正常に実行されているかどうかを確認するために、変数の値を表示する必要がある場合があります。CPU が停止している場合、「Watch」ウィンドウ内に変数を表示することができます。「Watch」ウィンドウを表示するには、「View」→「Watch Window」の順に選択します。「Watch Locals」タブには、現在実行中の関連する変数がすべて表示されます。

while ループにステップインし続けると、変数の値がプログラムの実行毎に変化する様子を観察できます。また、特定の変数の値を表示するには、マウス・ポインタを変数に合わせるか、「Watch1」タブに変数を置きます。変数と「Watch」ウィンドウに関する詳細は、5.2.4 項を参照してください。

2.5.5 「Output」ウィンドウ

「Output」ウィンドウは、デフォルトでは画面の一番下に表示されます。このウィンドウには、「View」→「Output Window」の順に選択してもアクセスできます。デフォルトでは、printf 関数により、この「Output」ウィンドウが生成されます。この「Output」ウィンドウには、Stdout の内容およびビルド時のログなどの情報が表示されます。

2.5.6 シンボル・ブラウザ

シンボル・ブラウザを使用すると、プロジェクト内のすべての構成要素を 1 度クリックするだけでアクセスできます。このウィンドウを表示するには、「View」→「Symbol Browser」の順に選択します。シンボル・ブラウザには、複数のタブ（「Files」、「Functions」および「Globals」）があります。

「Files」タブ内にあるツリーを展開すると、プロジェクト内のソース・ファイルが表示されます。「Files」タブまたは「Functions」タブ内のファイルをダブルクリックすると、自動的にファイルにアクセスできます。「Globals」タブでは、プロジェクト内のグローバル・シンボルにアクセスできます。

シンボル・ブラウザに関する詳細は、5.2.9 項を参照してください。

ここまでのステップを実行すると、Code Composer Studio を使用した初めてのプログラムの作成、ビルド、ロード、デバッグを首尾よく行うことができるようになっています。

2.6 ヘルプについて

Code Composer Studio では、「Help」メニューから多数のヘルプ・ツールにアクセスできます。内容で検索するには、「Help」→「Contents」の順に選択します。Code Composer Studio の開発プロセスの各チュートリアルをアクセスするには、「Help」→「Tutorial」の順に選択します。

最新のヘルプ・トピックおよび他のガイドを取得するには、「Help」→「Web Resources」の順に選択します。各機能またはプロセス関連の情報が記述されているユーザー・マニュアルは、PDF ファイルになっています。

更新情報および多数のオプションのプラグインは、「Help」→「Update Advisor」からアクセスできます。

第 3 章

ターゲットとホストのセットアップ

この章では、シングル・プロセッサとマルチプロセッサの両方の構成に合わせてターゲット構成を定義し、セットアップする方法、およびいくつかの汎用 IDE オプションを目的に合わせてカスタマイズする方法について説明します。

項目 **ページ**

3.1 ターゲットのセットアップ

3.1.1 Code Composer Studio のセットアップ・ユーティリティ

この章では、シングル・プロセッサとマルチプロセッサの両方の構成に合わせてターゲット構成を定義し、セットアップする方法、およびいくつかの汎用 IDE オプションをカスタマイズする方法について説明します。

3.1.1.1 既存の構成の追加

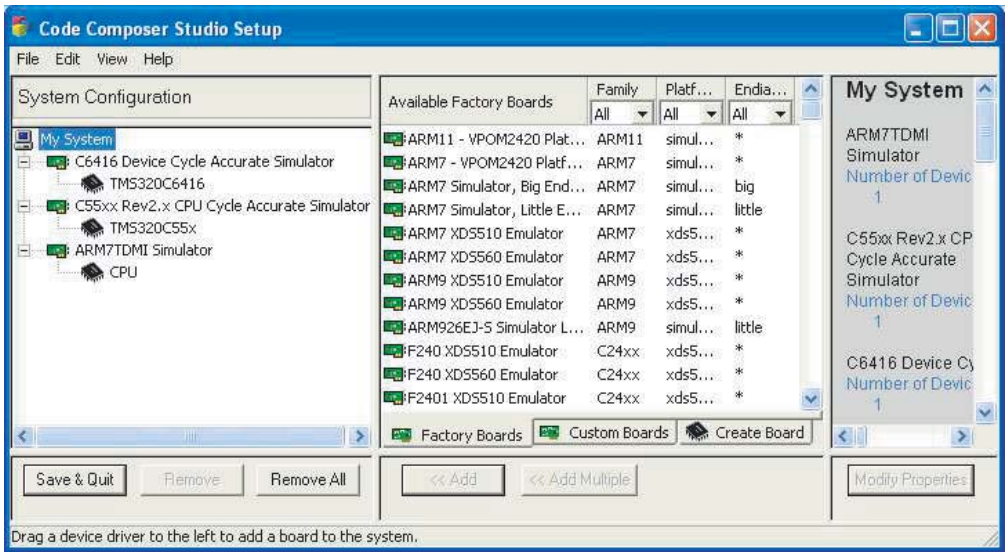
Setup ユーティリティを使用すると、異なるハードウェアまたはシミュレータのターゲットで動作するようにソフトウェアを構成できます。Code Composer Studio IDE を起動する前に、Setup ユーティリティで特定の構成を選択しておく必要があります。

付属している標準構成ファイルを使用して構成を作成することも、独自の構成ファイルを使用してカスタマイズした構成を作成することも可能です（オンライン・ヘルプまたはチュートリアルを参照）。ここでは、標準構成ファイルを使用します。

標準構成ファイルを使用して、システム構成を作成する手順は、次のとおりです。

1. デスクトップ上の「Setup Code Composer Studio」アイコンをダブルクリックします。「System Configuration」ダイアログ・ボックスが表示されます。
2. 「Available Factory Boards」のリストから、システムに適合する標準構成を選択します。システムに適合する構成があるかどうかを判断してください。適切な構成がない場合は、カスタマイズした構成を作成する必要があります（オンライン・ヘルプまたはチュートリアルを参照してください）。

図 3-1. 標準セットアップ構成



3. 「Add」ボタンをクリックして、現在作成しているシステム構成に選択した項目をインポートします。選択した構成が、「Setup」ウィンドウの「System Configuration」区画にある「My System」アイコンの下に表示されます。
   構成に複数のターゲットがある場合、それぞれのボードに対してこれらの手順を繰り返します。
4. 「Save & Quit」ボタンをクリックして、構成を保存します。
5. 「Yes」ボタンをクリックして、作成した構成で Code Composer Studio IDE を起動します。これで、プロジェクトを開始することができます。プロジェクトの開始については、本書の第 4 章、またはオンライン・ヘルプとチュートリアルを参照してください。

3.1.1.2 新しいシステム構成の作成

新しいシステム構成をセットアップするには、「Code Composer Studio Setup」ダイアログ・ボックスを使用します。

「File」メニューから「Remove All」を選択して、未設定の構成から開始します （また、目的のシステム構成に近い標準構成またはインポートされた構成から開始することもできます。この場合、構成をロード後に下記のステップ 3 から開始してください)。

1. 「System Configuration」区画から、「My System」アイコンを選択します。
2. 「Available Factory Boards」区画から、使用するシステムを表すターゲット・ボードまたはシミュレータを選択します。マウスを使用して、My System の下の左側の画面に必要なボードをドラッグするか、「Add」ボタンをクリックします。目的のボードを見つけるために、ファミリー毎に、プラットフォーム毎に、エンディアン毎にボードのリストをフィルタリングできます。必要に応じて、My System の下の左側の区画に、複数のボードをドラッグできます。
3. 「Available Factory Boards」区画に目的のターゲット・ボードまたはシミュレータが表示されていない場合は、適合したデバイス・ドライバをインストールする必要があります（たとえば、サードパーティのベンダが提供するデバイス・ドライバを使用するか、以前のバージョンの Code Composer Studio のドライバを使用します)。「Installing/Uninstalling Device Drivers」(「Help」→ Contents」→「Code Composer Studio Setup」→「How To Start」→「Installing/Uninstalling Device Drivers」の順に選択）に進みます。次に、この項で説明されている作業を継続して、システム構成を完成させます。
4. 追加したプロセッサ・タイプをクリックし、次に示すいずれかの手順を使用して、「Connection Properties」ダイアログ・ボックスを開きます。
   - 「System Configuration」区画内にあるプロセッサ・タイプを右クリックし、コンテキスト・メニューから「Properties」を選択します。現在のプロセッサを選択している場合、「Properties」を選択すると、「Processor Properties」ダイアログ・ボックスが表示されます。
   - 「System Configuration」区画内にあるプロセッサ・タイプを選択してから、右側の区画内にある「Modify Properties」ボタンを選択します。
5. 「Connection Properties」ダイアログ・ボックスの「Connection Name & Data File」タブと「Connection Properties」タブに含まれる情報を編集します。
6. スタート時の GEL ファイル、Master/Slave 値、Startup モード、および BYPASS 名、およびビット数は、「Processor Properties」ダイアログ・ボックス内に含まれています。「Processor Properties」ダイアログ・ボックスにアクセスするには、目的のプロセッサ上で右クリックし、コンテキスト・メニューから「Properties」を選択します。プロセッサによっては、他のプロパティも使用できる場合があります。シミュレータを構成する際には、多くのプロパティがプロセッサに基づくデフォルトの値と共に表示されることがあります。
   「Connection Properties」および「Processor Properties」ダイアログ・ボックスには、異なるフィールドを含むタブがあります。表示されるタブおよび編集可能なフィールドは、選択したボードまたはプロセッサによって異なります。各タブ内で情報を入力したら、「Next」ボタンをクリックして次のタブに進むか、次のタブ自体をクリックします。完了したら、「Finish」ボタンをクリックします。

「Connection Properties」または「Processor Properties」ダイアログ・ボックスの構成方法の詳細は、オンライン・ヘルプを参照してください（「Help」→「Contents」→「Code Composer Studio Setup」→「Custom Setup」の順に選択)。

3.1.1.3 マルチプロセッサ用の構成の作成

1 つの CPU に 1 つのシミュレータまたは 1 つのターゲット・ボードという構成が最も一般的な構成です。ただし、次の方法でさらに複雑な構成を作成することができます。

- 複数のエミュレータをコンピュータに接続し、それぞれに独自のターゲット・ボードを設定します。

- ボード上のスキャン・パスをリンクする特別なハードウェアを使用して、複数のターゲット・ボードを 1 つのエミュレータに接続します。
- 複数の CPU を 1 つのボード上に作成します。これらの CPU は、すべて同じ種類にすることも異なるタイプにすることもできます（DSP、マイクロコントローラなど）。

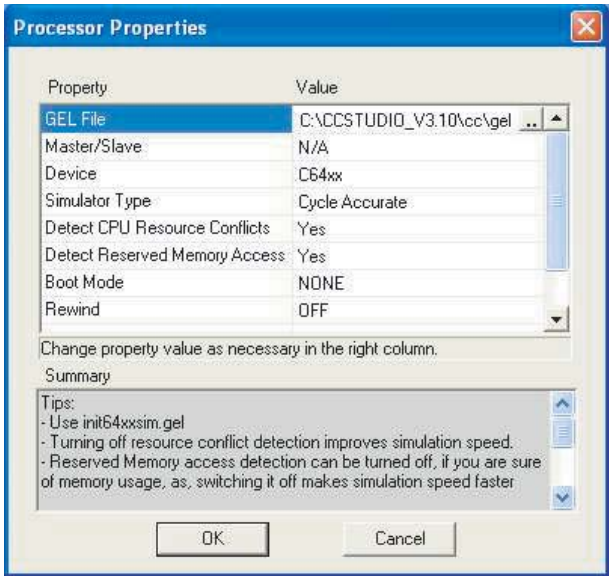
Code Composer Studio の構成は、一連の複数のボードとして表されますが、実際には各ボードは 1 つの CPU シミュレータか、マルチプロセッサを搭載する 1 つまたは複数のボードに接続できる 1 つのエミュレータ・スキャン・チェーンです。ボードに関連付けられているデバイス・ドライバは、そのスキャン・チェーン上にある CPU をすべて認識できる必要があります。詳細は、オンライン・ヘルプを参照してください（「Help」→「Contents」→「Code Composer Studio Setup」→「How To Start」→「Configuring CCStudio for Heterogeneous Debugging」の順に選択）。

3.1.1.4 スタートアップ GEL ファイル

GEL（General Extension Language; 汎用拡張言語）は、インタープリタ型言語で、C 言語によく似ています。GEL 関数は、Code Composer Studio の開発環境を構成するために使用できます。また、ターゲット CPU の初期化にも使用できます。さまざまな組み込み GEL 関数が使用可能です。また、ユーザー定義 GEL 関数を独自に作成することもできます。

「Processor Properties」ダイアログ・ボックスにある「GEL File」フィールドを使用すると、GEL ファイル（.gel）をシステム構成内の各プロセッサに関連付けることができます。「Processor Properties」ダイアログ・ボックスにアクセスするには、現在のプロセッサを選択し、コンテキスト・メニューから「Properties」を選択します。

図 3-2. GEL ファイル構成



Code Composer Studio が起動すると、スタートアップ GEL ファイルがスキャンされ、このファイルに含まれている GEL 関数がすべてロードされます。GEL ファイルに StartUp() 関数が含まれている場合、その関数内のコードも実行されます。たとえば、GEL マッピング関数を使用すると、デバッガに対してプロセッサのメモリを記述するメモリ・マップを作成できます。

```
StartUp(){ /*Everything in this function will be executed
 on startup*/  GEL_MapOn();  GEL_MapAdd(0, 0, 0xF000, 1,
 1); GEL_MapAdd(0, 1, 0xF000, 1, 1);}
```

GEL ファイルは、非同期であって、同期ではありません。つまり、GEL ファイル内にあるコマンドは、先行のコマンドの実行が完了する前に実行されます。詳細は、Code Composer Studio のオン

ライン・ヘルプを参照してください （「Help」→「Contents」→「Creating Code and Building Your Project」→「Automating Tasks with General Extension Language」の順に選択）。

3.1.1.5 デバイス・ドライバ

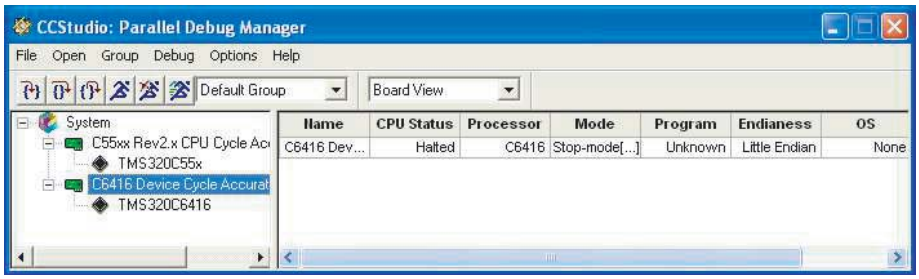
デバイス・ドライバと呼ばれる特殊なソフトウェア・モジュールを使用して、ターゲットと通信します。それぞれのドライバ・ファイルには、特定のターゲット構成が定義されています （ターゲット・ボードとエミュレータ、またはシミュレータ）。デバイス・ドライバは、TI またはサードパーティのベンダから入手できます。

「Available Factory Boards」区画内に表示される各ターゲット・ボードまたはシミュレータは、物理的にはデバイス・ドライバ・ファイルです。Code Composer Studio IDE は、デバイス・ドライバの作成をサポートしていませんが、TI またはサードパーティでは、インストール済みのドライバとは別にデバイス・ドライバを提供する場合もあります。

3.1.2 パラレル・デバッグ・マネージャ

マルチプロセッサ構成で、Code Composer Studio を起動すると、パラレル・デバッグ・マネージャ・プラス（PDM+）という特別な制御が開始されます。

図 3-3. パラレル・デバッグ・マネージャ



パラレル・デバッグ・マネージャを使用すると、ターゲット・デバイス毎に、Code Composer Studio IDE セッションを別々に開くことができます。PDM による制御を使用して、指定したデバイス上での動作を並列に制御できます。

パラレル・デバッグ・マネージャ・プラス（PDM+）の旧バージョンからこのバージョンに加えられた変更点は、次のとおりです。

- PDM+ の右側の区画にあるプロセッサを右クリックして、ターゲットの接続または切断を「すぐに」行うことができます。
- インターフェイスでは、プロセッサのビューを拡張したり、ドロップダウンのフィルタを使用してリストをグループ毎、CPU 毎、またはボード毎に表示することができます。
- 左側の区画にあるプロセッサ・アイコンが赤で強調表示されている場合、プロセッサがシステムに接続されていないか、ステータス情報が更新されていることを示します。
- プロセッサを疎結合グループ（プロセッサがすべて同一物理スキャン・チェーン上にあるわけではない）に入れることができます。ツールバー上にある 2 番目のドロップダウン・メニューから「Group View」と、PDM+ の左側の区画にある「System」を選択すると、同期グループと非同期グループを表示できます。

グローバル・ブレークポイントが動作するのは、グループ内のプロセッサが同一物理スキャン・チェーンに属している場合のみです。

パラレル・デバッグ・マネージャの詳細は、オンライン・ヘルプ（「Help」→「Contents」→「Debugging」→「Parallel Debug Manager」の順に選択）を参照してください。

3.1.3 接続 / 切断

Code Composer Studio IDE では、接続 / 切断という新機能を使用して、ターゲットとの動的な接続または切断を容易に行うことができます。接続 / 切断を使用すると、ハードウェア・ターゲットから切断できるだけでなく、再度接続する際に前回のデバッグ状態を復元することができます。

制御ウィンドウが開いている場合、デフォルトでは、Code Composer Studio IDE はターゲットに接続しようとしません。ターゲットへの接続を確立するには、「Debug」→「Connect」の順に選択します。デフォルトの動作は、「Option」→「Customize」の順に選択して表示される「Debug Properties」タブ内で変更できます。

ステータス・バーでは、ターゲットの状態の変更を示すヘルプ・アイコンが少しの間点滅します。ターゲットが切断されると、ステータス・バーにこの情報が表示され、ターゲットの最終的な実行状態（停止、実行中、自由動作、エラー状態など）が示されます。接続時に、ステータス・バーには、ターゲットがステップ実行（イン、オーバー、アウト）されているかどうかも示されます。また、停止状態を引き起こしたブレークポイントのタイプ（ソフトウェアまたはハードウェア）も示されます。

ターゲットへの接続後（最初の接続を除く）、「Debug」メニューの下にある「Restore Debug State」というメニュー・オプションが使用できるようになります。このオプションを選択すると、切断時に無効になったブレークポイントがすべて有効になります。これらのブレークポイントをリセットするには、F9 を押すか、右クリック・メニューから「Toggle Breakpoints」を選択します。この操作を行っても、cTools ジョブとエミュレータ解析からのブレークポイントは有効にはなりません。

パラレル・デバッグ・マネージャが開いている場合には、「Name」という列の下にあるターゲット・デバイスに対応するセル上で右クリックして、ターゲットに接続することができます。

接続 / 切断に関する詳細は、Code Composer Studio のオンライン・ヘルプを参照してください（「Debugging」→「Connect/Disconnect」の順に選択）。

3.2 ホスト IDE のカスタマイズ

Code Composer Studio が正常に構成され、IDE が起動すると、使用する目的に合わせていくつかの汎用 IDE オプションをカスタマイズできます。

3.2.1 デフォルトの色とフォント

「Option」→「Font」→「Editor Font」および「Option」→「Color」→「Editor Color」の順に選択すると、CodeWright テキスト・エディタ（4.2.2 項を参照）のデフォルトの外観を変更することができます。

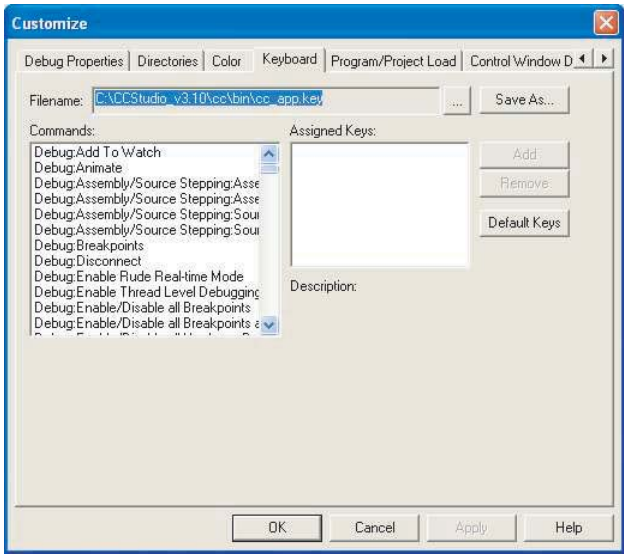
「Option」→「Font」→「Tools Font」および「Option」→「Color」→「Tools Color 」の順に選択すると、各種 IDE ツールのウィンドウのデフォルトの外観を変更することができます。

3.2.2 デフォルトのキーボード・ショートカット

IDE にはデフォルトで、変更可能な 80 以上の事前定義済みのキーボード・ショートカットが組み込まれています。ドキュメント・ウィンドウから起動される編集コマンドまたはデバッグ・コマンドに対応するキーボード・ショートカットを新規に作成することもできます。キーボード・ショートカットを割り当てる手順は、次のとおりです。

1. 「Option」→「Customize」の順に選択します。

図 3-4. キーボード・ショートカットの変更



2. 「Customize」ダイアログ・ボックスから、「Keyboard」タブを選択して、次のオプションを表示します。

   - **Filename :** デフォルトでは、標準のキーボード・ショートカット・ファイルが表示されます。保存されているキーボード構成ファイル（*.key）をロードするには、ファイルのパスと名前を入力するか、参照ボタン（...）を使用してファイルを指定します。
   - **Commands :** キーボード・ショートカットに割り当てるコマンドを選択します。
   - **Assigned Keys** : 選択したコマンドに割り当てられているキーボード・ショートカットが表示されます。
   - **Add :**「Add」ボタンをクリックすると、選択したコマンドを呼び出す新しいキー・シーケンスを割り当てることができます。「Assign Shortcut」ダイアログ・ボックスに新しいキー・シーケンスを入力し、「OK」をクリックします。
   - **Remove :** コマンド用の特定のキー・シーケンスを削除するには、「Assigned Keys」リストから削除するキー・シーケンスを選択し、「Remove」ボタンをクリックします。
   - **Default Keys :** 「Default Keys」ボタンをクリックすると、デフォルトのキーボード・ショートカットにすぐに戻すことができます。
   - **Save As :** 「Save As」ボタンをクリックすると、カスタマイズしたキーボード構成をファイルに保存できます。「Save As」ダイアログ・ボックスで、構成を保存する場所を指定し、キーボード構成ファイルに名前を付け、「Save」をクリックします。

3. 「OK」をクリックして、ダイアログ・ボックスを閉じます。

3.2.3 他の IDE のカスタマイズ

- 「Option」→「Customize」→「File Access」の順に選択して、「File」メニューや「Project」メニューの「Recent...」で表示される最近使用したファイルやプロジェクトの数を指定できます。
- 「Option」→「Customize」→「Control Window Display」の順に選択して、タイトル・バー内に表示される情報（プロセッサ・タイプ、プロジェクト名、パスなど）を設定できます。

- 「Option」→「Customize」→「Control Window Display」の順に選択して、ウィンドウが閉じられるときのデフォルトのオプションを設定できます。プロジェクトを閉じると、IDE が自動的にそのプロジェクトに関するすべてのウィンドウを閉じるように指定することもできます。または、制御ウィンドウを閉じると常にすべてのプロジェクトを閉じるように選択することもできます。
- CodeWright を使用する「Code」ウィンドウをカスタマイズできます（4.2.2 項を参照）。

第 4 章

コードの作成

この章では、コードの作成および基本的な Code Composer Studio IDE プロジェクトのビルドに適用可能なオプションについて説明します。

項目 **ページ**

4.1 プロジェクトの構成

プロジェクトには、個々のプログラムまたはライブラリのビルドに必要な情報がすべて保存されています。

- ソース・コードおよびオブジェクト・ライブラリのファイル名
- コード生成ツールのオプション
- インクルード・ファイルの依存関係

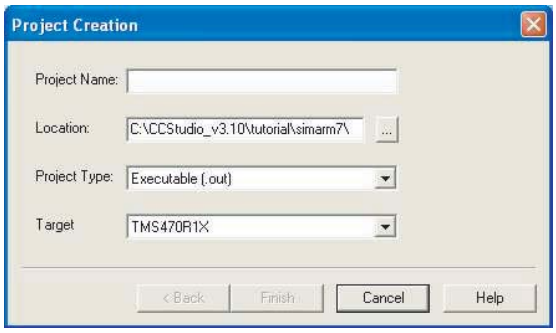
4.1.1 プロジェクトの作成

1 つまたは複数のプロジェクトを新規に作成する手順は、次のとおりです（複数のプロジェクトを同時に開くことができます)。プロジェクトのファイル名は、固有でなければなりません。

プロジェクトの情報は、1 つのプロジェクト・ファイル（*.pjt）に保存されます。

1. 「Project」メニューから、「New」を選択します。「Project Creation」ウィザード・ウィンドウが表示されます。

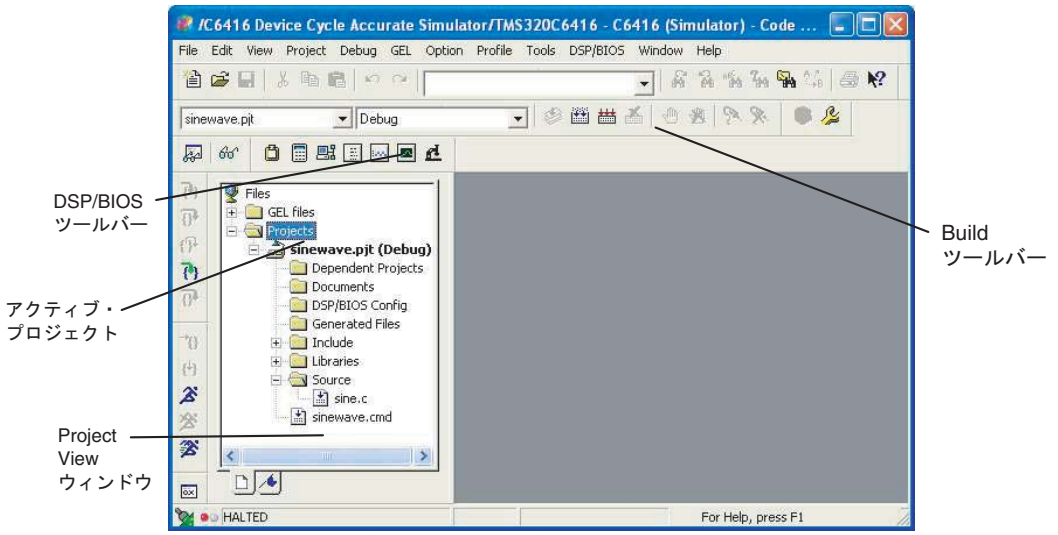
図 4-1. プロジェクト作成ウィザード



2. 「Project Name」フィールドに、プロジェクト名を入力します。
3. 「Location」フィールドで、プロジェクト・ファイルを保存するディレクトリを指定します。コンパイラとアセンブラで生成されたオブジェクト・ファイルもこのディレクトリに保存されます。「Location」フィールドにフルパスを入力するか、参照ボタン（...）をクリックしたときに表示される「Choose Directory」ダイアログ・ボックスを使用します。新規プロジェクト毎に異なるディレクトリを使用することをお勧めします。
4. 「Project Type」フィールドで、ドロップダウン・リストからプロジェクト・タイプを選択します。Executable (.out) または Library (lib) を選択します。Executable は、実行可能なファイルを生成するプロジェクトを示します。Library は、オブジェクト・ライブラリを生成するプロジェクトを示します。
5. 「Target」フィールドで、使用する CPU を示すターゲット・ファミリーを選択します。この情報は、複数のターゲット用にツールがインストールされている場合に必要です。
6. 「Finish」をクリックします。プロジェクト名 .pjt というプロジェクト・ファイルが作成されます。このファイルには、プロジェクトで使用されるすべてのファイルとプロジェクトの設定内容が保存されています。

新規プロジェクトと最初のプロジェクト構成（アルファベット順）がアクティブなプロジェクトになり、TI が提供するデバッグ構成用とリリース構成用のデフォルトのコンパイラとリンカのオプションが継承されます。

図 4-2. CCStudio IDE 制御ウィンドウ



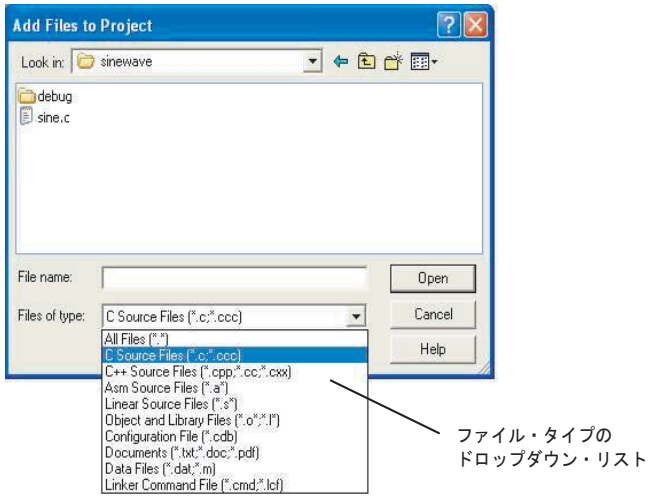
新規プロジェクト・ファイルを作成してから、ソース・コード、オブジェクト・ライブラリ、リンカ・コマンドのファイル名をプロジェクト・リストに追加します。

4.1.1.1 プロジェクトへのファイルの追加

複数の異なるファイルまたはファイル・タイプをプロジェクトに追加できます。これらのタイプを次の図に示します。プロジェクトにファイルを追加する手順は、次のとおりです。

1. 「Project」→「Add Files to Project」の順に選択するか、「Project View」ウィンドウ内でプロジェクトのファイル名を右クリックして「Add Files to Project」を選択します。「Add Files to Project」ダイアログ・ボックスが表示されます。

図 4-3. 「Add Files to Project」ダイアログ・ボックス



2. 「Add Files to Project」ダイアログ・ボックスで、追加するファイルを指定します。現在のディレクトリにファイルが表示されていない場合、目的の場所に移動します。「Files of Type」ドロップダウン・リストを使用して、「File name」フィールドに表示されるファイルのタイプを指定します。

注：
ヘッダ / インクルード・ファイル（*.h）を手作業でプロジェクトに追加しないでください。これらのファイルは、ビルド・プロセス中にソース・ファイルの依存関係がスキャンされる際に自動的に追加されます。

3.「Open」をクリックして、指定したファイルをプロジェクトに追加します。

ファイルが現在のプロジェクトに追加されると、「Project View」ウィンドウ（図 4-2 を参照）は自動的に更新されます。

プロジェクト・マネージャはファイルをソース・ファイル、インクルード・ファイル、ライブラリ、および DSP/BIOS 構成ファイルの各フォルダに整理します。DSP/BIOS で生成されるソース・ファイルは、「Generated Files」フォルダに格納されます。Code Composer Studio IDE は、プログラムをビルドする際に、次の順序でプロジェクト・ファイルを検索します。

- ソース・ファイルが含まれているフォルダ。
- コンパイラまたはアセンブラのオプションの「Include search path」に指定してあるフォルダ（左から右）。
- オプションの環境変数 DSP_C_DIR（コンパイラ）と DSP_A_DIR（アセンブラ）の定義の中で指定されているフォルダ（左から右）。

4.1.1.2 ファイルの削除

プロジェクトからファイルを削除するには、「Project View」ウィンドウ内のファイルを右クリックし、コンテキスト・メニューから「Remove from Project」を選択します。

4.1.2 プロジェクト構成

プロジェクト構成では、プロジェクト・レベルのビルド・オプションを定義しています。このレベルで指定されたオプションは、プロジェクト内のすべてのファイルに適用されます。

プロジェクト構成を用いると、プログラム開発の異なるフェーズに対してビルド・オプションを定義できるようになります。たとえば、プログラムのデバッグ時に使用する Debug 構成、および完成した製品をビルドするために使用する Release 構成を定義することができます。

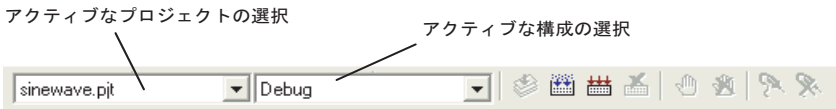
各プロジェクトには、2 つのデフォルト構成 Debug と Release が作成されます。さらに追加の構成を定義することができます。プロジェクトを作成するか、または既存のプロジェクトを最初に開くときは必ず、最初の構成（アルファベット順）がワークスペース内でアクティブに設定されます。

プログラムをビルドする時、ソフトウェア・ツールで生成される出力ファイルはプロジェクト構成定義に基づくサブディレクトリに格納されます。たとえば、プロジェクトを MyProject ディレクトリに作成した場合、Debug 構成の出力ファイルは MyProject¥Debug ディレクトリに格納されます。同様に、Release 構成の出力ファイルは MyProject¥Release ディレクトリに格納されます。

4.1.2.1 アクティブなプロジェクト構成の変更

「Project」ツールバー内の「Select Active Configuration」フィールドをクリックし、ドロップダウン・リストから構成を選択します。

図 4-4. 構成ツールバー

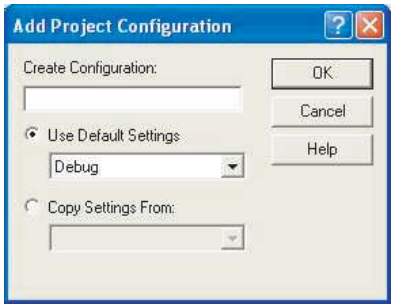


4.1.2.2 新しいプロジェクト構成の追加

1. 「Project」→「Configurations」の順に選択するか、「Project View」ウィンドウ内でプロジェクトのファイル名を右クリックして「Configurations」を選択します。

2. 「Project Configurations」ダイアログ・ボックスで、「Add」をクリックします。「Add Project Configuration」ウィンドウが表示されます。

図 4-5. 「Add Project Configurations」ダイアログ・ボックス



3. 「Add Project Configuration」ダイアログ・ボックスで、「Create Configuration」フィールドに新しい構成の名前を指定し、デフォルトの設定を使用する「Use Default Settings」（ビルド・オプション）、または既存の構成からコピーする「Copy Settings From」を選択して、新規構成を設定します。
4. 「OK」をクリックして選択項目を受け入れ、「Add Project Configuration」ダイアログ・ボックス閉じます。
5. 「Done」をクリックして、「Project Configurations」ダイアログ・ボックスを閉じます。
6. 「Project」メニュー内にある「Build Options」ダイアログ・ボックスを使用して、新規構成を変更します。

4.1.3 プロジェクトの依存関係

プロジェクトの依存関係ツールを使用すると、さらに複雑なプロジェクトの管理およびビルドを容易に行うことができます。プロジェクトの依存関係を使用すると、大きいプロジェクトを複数の小さいプロジェクトに分割することができます。また後からこれらのプロジェクト間の依存関係を使用して最終プロジェクトを作成することができます。サブプロジェクトは必ず最初にビルドされます。これは、メイン・プロジェクトがこれらのサブプロジェクトに依存するためです。

4.1.3.1 プロジェクトの依存関係の作成（サブプロジェクト）

プロジェクトの依存関係またはサブプロジェクトを作成するには、次の 3 つの方法があります。

- **「Project View」ウィンドウからのドラッグ・アンド・ドロップ**
  サブプロジェクトをターゲット・プロジェクトのアイコンまたはターゲット・プロジェクトの下にある「Dependent Projects」アイコンにドロップします。同じ「Project View」ウィンドウ内でドラッグ・アンド・ドロップするか、同時に実行する 2 つの Code Composer Studio の「Project View」ウィンドウ間でドラッグ・アンド・ドロップすることができます。
- **Windows エクスプローラからのドラッグ・アンド・ドロップ**
  1. Code Composer Studio 内にあるメイン・プロジェクトを開きます。
  2. Windows エクスプローラを起動します。エクスプローラと Code Composer Studio の両方が開いている状態にします。
  3. Windows エクスプローラで、サブプロジェクトにするプロジェクトの .pjt ファイルを選択します。
  4. この .pjt ファイルを Code Composer Studio の「Project Window」ウィンドウにドラッグします。移動しようとしている .pjt ファイルには、＋（プラス）記号が表示されます。
  5. このファイルをメイン・プロジェクトの「Dependent Projects」フォルダにドロップします。
- **コンテキスト・メニューを使用**
  「Project View」ウィンドウで、ロードされているプロジェクトの下にある「Dependent Projects」

アイコンを右クリックし、コンテキスト・メニューから「Add Dependent Projects」を選択します。表示されるダイアログ・ボックスで、他の .pjt プロジェクト・ファイルを参照し選択します。選択した .pjt ファイルが、ロードされているプロジェクトのサブプロジェクトになります。選択した .pjt ファイルがまだロードされていない場合は、自動的にロードされます。

4.1.3.2 プロジェクトの依存関係の設定

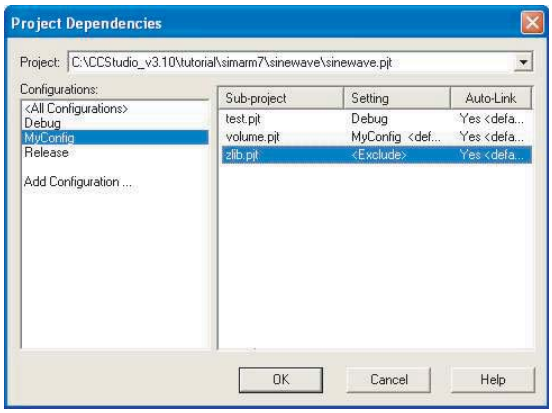
サブプロジェクトには、それぞれ独自の構成設定があります。さらに、メイン・プロジェクトには、各サブプロジェクト用の構成設定があります。これらの設定には、「Project Dependencies」ダイアログ・ボックスからアクセスできます。ダイアログ・ボックスを開くには、「Project」メニューから、またはプロジェクトのコンテキスト・メニューから「Project Dependencies」を選択します。

4.1.3.3 プロジェクト構成の変更

「Project Dependencies」ダイアログ・ボックスで、サブプロジェクトの設定を変更できます。前述のとおり、このダイアログ・ボックスにアクセスするには、「Project」→「Project Dependencies」の順に選択します。

図 4-6 に示すように、特定のサブプロジェクトを構成から除外することができます。次の例が示すように、sinewave.pjt の MyConfig 構成では zlib.pjt がビルドから除外されています。また、この構成に対して特定のサブプロジェクト構成を選択することもできます。MyConfig では、test.pjt がデフォルトの MyConfig サブプロジェクト構成ではなく、Debug 構成を使用してビルドされます。

図 4-6. 「Project Dependencies」ダイアログ・ボックス



4.1.3.4 サブプロジェクト構成

各サブプロジェクトには、独自のビルド構成があります。メイン・プロジェクト構成毎に、特定の構成を使用して各サブプロジェクトをビルドすることができます。サブプロジェクトの設定を変更するには、そのプロジェクトの横にあるドロップダウン・リストを使用します（Setting 列の下）。

4.1.4 メイクファイル

Code Composer Studio IDE では、外部メイクファイル（*.mak）、およびそれと関連付けられている外部 make ユーティリティを使用して、プロジェクト管理とビルド・プロセスのカスタマイズを行うことができます。

Code Composer Studio IDE でメイクファイルを使用してプログラムをビルドするには、そのメイクファイルを含む Code Composer Studio プロジェクトを作成する必要があります。Code Composer Studio プロジェクトをメイクファイルに関連付けると、プロジェクトとその内容が「Project View」ウィンドウに表示され、「Project」→「Build」コマンドと「Project」→「Rebuild All」コマンドを使用してプログラムをビルドできるようになります。

1. 「Project View」ウィンドウ内でメイクファイルの名前をダブルクリックし、編集するメイクファイルを開きます。
2. メイクファイルのビルド・コマンドとオプションを変更します。
   特別なダイアログ・ボックスを使用すると、メイクファイルのビルド・コマンドとメイクファイルのオプションを変更できます。メイクファイルを用いての作業では、通常の Code Composer Studio の「Build Options」ダイアログ・ボックスは使用できません。

複数の構成を作成し、それぞれに独自のビルド・コマンドとオプションを指定できます。

注：
制限と制約事項：「Project View」内ではソース・ファイルをプロジェクトに追加したり、プロジェクトから削除することができます。ただし、「Project View」内で変更を加えても、メイクファイルの内容は変更されません。これらのソース・ファイルは、ビルド・プロセスに影響を与えたり、メイクファイルの内容にも反映されません。同様に、メイクファイルを編集しても、「Project View」内の内容が変更されることはありません。「Project View」内で追加されるソース・ファイルのファイル固有のオプションは、無効となります。「Project」→「Compile File」コマンドも使用できません。ただし、プロジェクトが保存される時に「Project View」の現在の状態は保持されます。

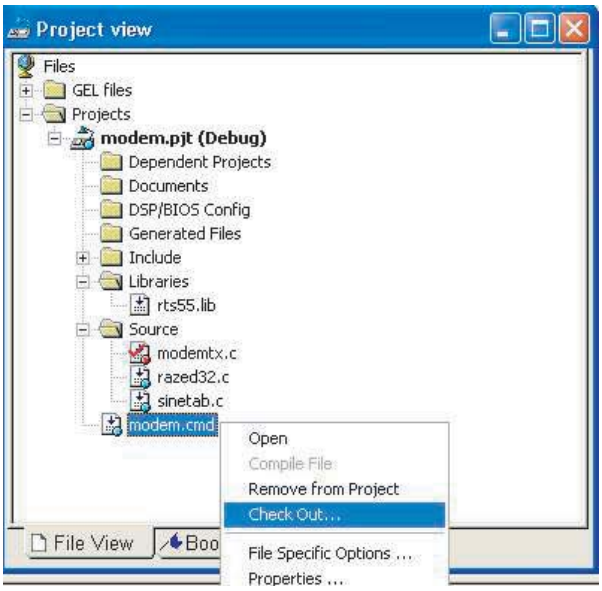
注：
Code Composer Studio IDE コマンドでメイクファイルを使用してプログラムをビルドするには、事前に必要な環境変数を設定しておく必要があります。環境変数を設定するには、次のバッチ・ファイルを実行します。
DosRun.bat
このバッチ・ファイルは、C:¥CCStudio_v3.1 ディレクトリ内にあります。Code Composer Studio IDE を C:¥CCStudio_v3.1 以外のディレクトリにインストールした場合は、バッチ・ファイルはインストール時に指定したディレクトリ内にあります。

4.1.5 ソース・コントロールの統合

プロジェクト・マネージャを使用すると、プロジェクトをさまざまなソース・コントロール・プロバイダに接続することができます。Code Composer Studio IDE では、インストールされている互換性のあるプロバイダがすべて自動的に検出されます。

1. 「Project」メニューから、「Source Control」を選択します。
2. 「Source Control」サブメニューから、「Select Provider」を選択します。
3. 使用するソース・コントロール・プロバイダを選択し、「OK」をクリックします。
   ソース・コントロール・プロバイダが表示されていない場合、プロバイダのクライアント・ソフトウェアがコンピュータに正しくインストールされているかどうかを確認してください。
4. プロジェクトを 1 つ開き、「Add to Source Control from Project」→「Source Control」の順に選択します。
5. ソース・ファイルをソース・コントロールに追加します。
6. ファイルがソース・コントロールの管理下にあるかどうかをチェックするには、「Project View」ウィンドウ内でファイルを選択し、そのファイルを右クリックします。アイコンは、ソース・コントロールに接続されているソース・ファイルを識別します。

図 4-7. ソース・コントロールの統合

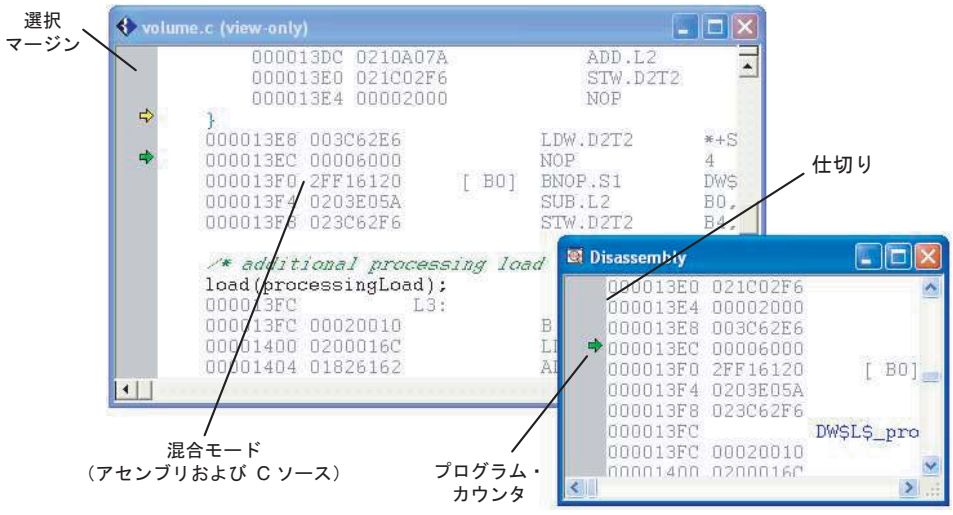


4.2 テキスト・エディタ

4.2.1 コードの表示と編集

「Project View」内にあるファイル名をダブルクリックすると、「IDE」ウィンドウ内にソース・コードが表示されます。

図 4-8. ソース・コード・ウィンドウの要素



- **選択マージン**
  デフォルトでは、選択マージンは内蔵エディタと「Disassembly」ウィンドウの左側に表示されます。選択マージン内の色付きアイコンは、この位置にブレークポイント（赤）またはプローブポイント（青）が設定されていることを示します。黄色の矢印は、プログラム・カウ

ンタ（PC）の位置を示します。選択マージンのサイズは、仕切りをドラッグして変更することができます。

- **キーワード**
  内蔵エディタには、キーワードを強調表示する機能があります。キーワード、コメント、文字列、アセンブラ擬似命令、GEL コマンドが別々の色で強調表示されます。さらに、キーワードのセットを新規に作成したり、デフォルトのキーワードのセットをカスタマイズして、キーワード・ファイル（*.kwd）に保存したりすることができます。
- **キーボード・ショートカット**
  デフォルトのキーボード・ショートカットを変更できます。キーボード・ショートカットを新規に作成して、ドキュメント・ウィンドウから編集コマンドまたはデバッグ・コマンドを起動できるようにすることもできます。キーボード・ショートカットを変更するには、「Option」メニューの「Customize」ダイアログ・ボックスを使用します。
- **ブックマーク**
  ブックマークを使用すると、ソース・ファイル内の重要な位置を検索して保持できます。ブックマークは、ソース・ファイル内の任意の行に設定できます。

4.2.2 「Code」ウィンドウのカスタマイズ

IDE に組み込まれているテキスト・エディタ（CodeWright と呼ばれる）では、コードの書式設定と動作をカスタマイズできます。「Option」→「Editor」メニューには、「Language」、「ChromaCoding Lexers」、「View Setups」に対応する追加オプションが含まれています。

- **Language**
  ファイル・タイプ（.cpp、.awk など）を動作のセットに関連付けることができます。ファイル・タイプのリストは、ChromaCoding Lexers のリストとは違うことに注意してください。デフォルトでは、ファイル・タイプの多くは、適切な字句解析器（lexer）に関連付けられています（つまり、.h ファイル・タイプは C 字句解析器に関連付けられている）。ファイル・タイプには、字句解析器にまったくマップされないものがあります。
- **ChromaCoding Lexers**
  字句解析器の辞書には、プログラミング言語を色付きで表示できる設定コレクションが保存されています。この字句解析器の辞書には、識別子、カッコ、プリプロセッサ、キーワード、演算子、文字列、コメントが含まれます。CodeWright テキスト・エディタには、Code Composer Studio IDE 用の字句解析機能（GEL、CCS、C、DSP/BIOS など）を含む、約 20 言語にもおよぶ言語特有の字句解析機能が使えるように設定されています。また、「ChromaCoding Lexer」ダイアログ・ボックスの右側にある「New」ボタンまたは「Save as」ボタンをクリックして、新規に字句解析機能を作成することもできます。
- **View Setups**
  これは、1 つのプログラミング言語に特有ではない、より汎用的な機能を指定します。たとえば、すべての言語のコメントは青で表示されるということを指定することができます。ただし、字句解析機能は、テキスト・エディタでコメントの前後に使用するコメントを表す区切り文字を指定します。

表 4-1. CodeWright テキスト・エディタ クイック・リファレンス

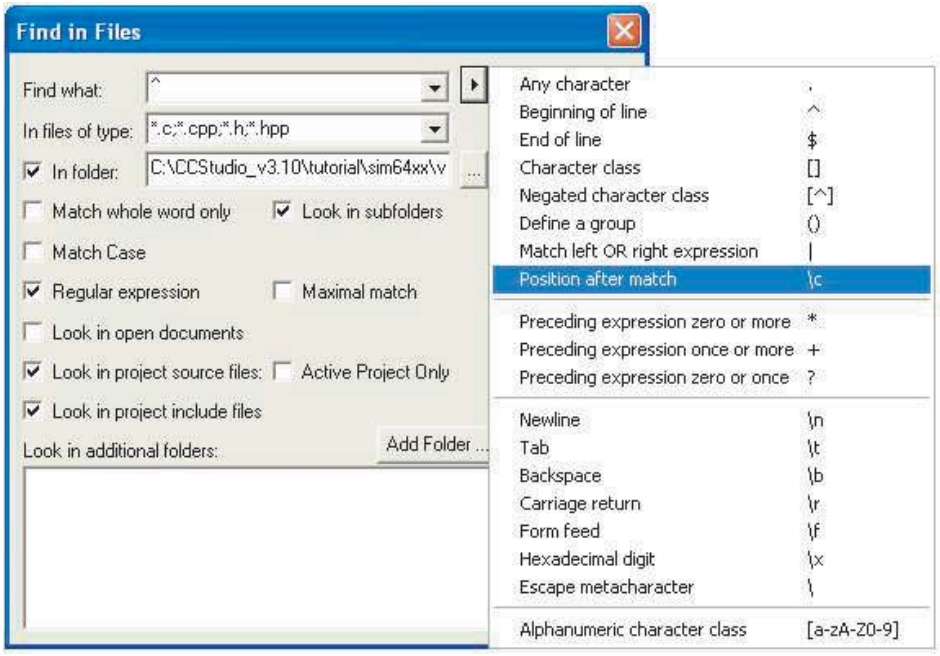
CodeWright のメニュー位置	構成可能な設定とオプション
エディタ属性（グローバル設定）： 「Option」→「Editor」→「Properties」	エディタ、ファイルのロード、デバッグ、選択マージンのサイズ変更、ツールヒント、外部エディタ、バックアップ（自動保存）のオプション
ファイル・タイプの設定（言語属性）： 「Option」→「Editor」→「Language」	言語オプションとマッピング、タブとインデント、テンプレート、コード・テキストのカラー設定、CodeSense、ファイル・タイプ毎の書式設定、コメント
字句解析機能の設定（言語特有の字句解析機能の設定）： 「Option」→「Editor」→「ChromaCoding Lexers」	識別子、カッコの文字、正規表現を除くカラー、新しい用語（キーワード、プリプロセッサ、演算子）の追加とキーワードのデフォルト、言語特有のコメント、文字列のデフォルト、数字の要素
View Setups（追加グローバル設定）： 「Option」→「Editor」→「View Setups」	行番号とルーラーの表示、行の強調表示、スクロール、行番号領域の幅、変数の表示（行末、タブ、スペースなど）、一般的なカラー・デフォルト、一般的なフォント・デフォルト
高度なテキスト処理： 「Edit」→「Advanced」の順に選択するか、テキスト・ウィンドウ内で右クリックしてから「Advanced」を選択	大文字から小文字への変換（その逆も）、コメントと関数、タブからスペースへの変換（その逆も）、他の高度な編集オプション

4.2.3 エディタのテキスト処理機能

テキスト・エディタには、テキストを処理するための機能が多数組み込まれています。

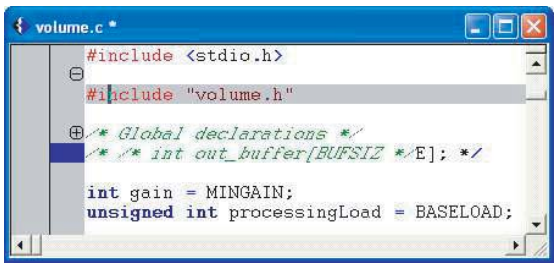
- **差分抽出とマージ機能**
  差分抽出機能（「File」→「Difference between files」）を使用すると、2 つの同じようなファイルを比較し、その違いを表示できます。マージ機能（「File」→「Merge Files」）を使用すると、複数のファイルをマージすることができます。
- **正規表現のサポート**
  「Edit」→「Find in Files」の順に選択するか、「Edit」→「Replace in Files」の順に選択します。テキスト・エディタでは、通常の検索または置換機能だけでなく、より複雑なテキスト処理を行うために正規表現を使用することもできます。たとえば、特定のディレクトリ内にあるすべてのファイルに対して、グローバル置換することができます。また、検索結果の保存機能を使用したり、ヘルパー・ドロップダウン・ウィンドウ（下記参照）を使用して、正規表現を構成しやすくすることができます。

図 4-9. テキスト・エディタでの正規表現の使用



- **コードの非表示と表示の切り替え**
  選択表示機能を使用すると、選択したパラメータに従って、特定のコードを表示したり、非表示にすることができます。たとえば、エディタが選択表示機能を使用して特定のコードを拡張したり、折り畳んだりすることを指定できます。また、適切なオプションを選択して、すべての関数定義またはプリプロセッサ擬似命令を非表示にすることもできます。この操作を行うと、コードが非表示にされたことを示す小さなアイコンがマージン部分に表示されます（図 4-10 を参照）。この小さなアイコンをクリックすると、特定のブロックにあるコードの表示と非表示を切り替えることができます。

図 4-10. 選択表示機能



4.2.4 自動保存のデフォルト設定

テキスト・エディタは、システムがクラッシュした場合に備えて定期的に作業ファイルを保存して、作業損失を防止するようにしています。この機能を使用するには、「Option」→「Editor」→「Properties」→「Backup」の順に選択し、Auto-save enable チェックボックスにチェックマークを付けます。また、ファイルを保存するまでの時間間隔を選択したり、バックアップ・ファイルの名前と保存先を指定することもできます。CCStudio は、バックアップ・ファイルを上書きする前に確認メッセージを表示します（ただし、メッセージが表示されないように指定した場合を除きます）。

4.2.5 自動補完、ツールヒント、変数ウォッチ（CodeSense）

CodeWright テキスト・エディタは、CodeSense と呼ばれる自動補完エンジンを使用しています。ツールヒントまたは自動補完機能がアクティブになっている場合、アイコンがコードのカーソル行の下に表示されます。アイコンは、C、C++、Java のコードに対する記号、関数パラメータ、ツールヒントを示します。また、ツールヒントは、変数ウオッチにも使用できます。

CodeSense DLL が有効になっているとき、CodeSense は特定のファイル・タイプとのみ機能します。

CodeSense を有効にする手順は、次のとおりです。

1. 「Option」→「Editor」→「Language」→「CodeSense」の順に選択します。
2. 左側のボックスで、連携させるファイル・タイプを強調表示します。
3. 「File Type」チェックボックスの右側で、CodeSense DLL が有効であることを確認します（CodeSense がファイル・タイプをサポートしていない場合、Code Sense DLL チェックボックスは無効になっています）。

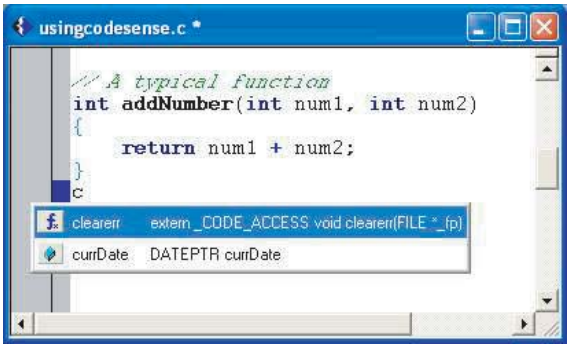
CodeSense DLL を有効にすると、CodeSense を次の目的で使用することができます。

- 入力中のシンボルに関連付けられているシンボル（関数、構造体、マクロ、メンバなど）をリストする。
- 現在編集中のドキュメントにコンテキスト・リストからシンボルを挿入して、入力中のシンボルを補完する。
- リスト内にある選択したシンボルの Goto ボタンを使用してシンボル定義をアクセスする（Ctrl-G は、Goto 機能に対応するキーボード・ショートカットです）。
- 入力中の関数に必要なパラメータをリストするツールヒントを取得する。
- 自動的に表示されるか、Ctrl または Shift のいずれかが押されたときに表示される（CodeSense の設定によって異なる）ホーバー・ツールヒント内にあるシンボルの定義を参照する。

CodeSense のワード補完機能は、シンボルの入力を完成するのに役立ちます。文字を一部入力してから、この機能を使用する手順は、次のとおりです。

1. Ctrl と Space を一緒に押し、選択対象となる状況依存のシンボルのリスト・ボックスを表示します。シンボルは、入力していた文字で始まり、右側の列には、各シンボルの定義が表示されます。

図 4-11. コード・センス



2. リストから適切なシンボルを選択して強調表示します。選択したシンボルの対応する画像（Goto ボタン）を押して、ライブラリのソース・コード内にあるシンボルの定義を表示します。また、キー・シーケンス Ctrl-G は、選択したシンボルの定義もアクセスします。
3. ドロップダウン・リストがまだ表示されている間に、Enter を押します。強調表示されたシンボルが自動的にソース・コードに入力され、入力中のワードを補完します。

4.2.6 外部エディタの使用

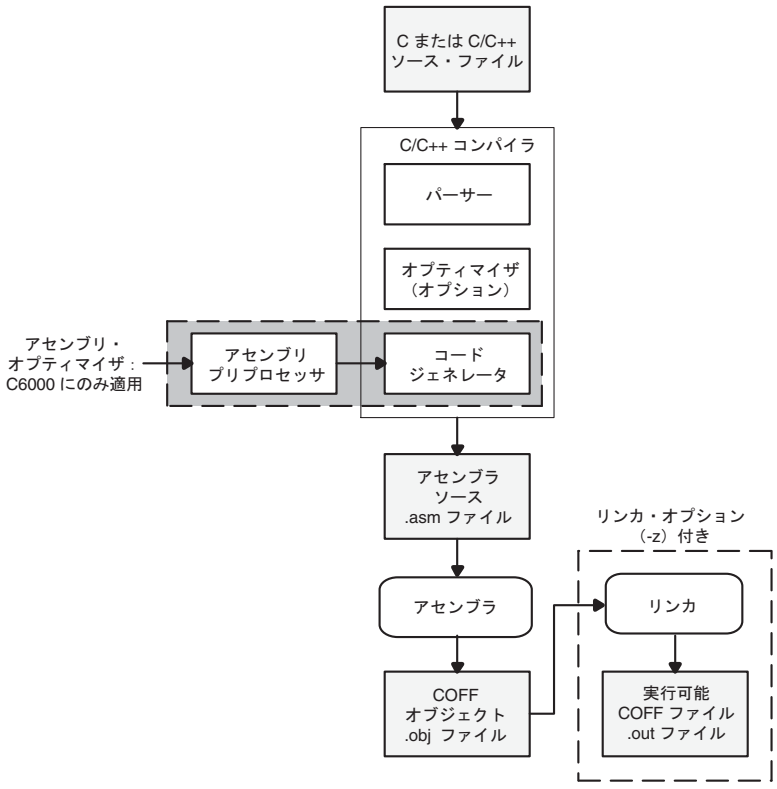
Code Composer Studio IDE では、デフォルトの内蔵エディタの代わりに、外部（サードパーティ製）のテキスト・エディタを使用できます。外部エディタを構成して有効にすると、新規文書を作成するか、既存ファイルを開くたびに外部エディタが起動します。外部エディタは、ファイルの編集作業にのみ使用できます。プログラムをデバッグするには、内蔵エディタを使用する必要があります。外部エディタを設定するには、「Option」→「Editor」→「Properties」の順に選択して表示されるダイアログ・ボックスから「External Editor」タブを選択して選択します。

4.3 コード生成ツール

4.3.1 コード開発フロー

コード生成ツールには、最適化 C/C++ コンパイラ、アセンブラ、リンカ、および各種ユーティリティが含まれています。次の図では、コード生成時にこれらのツールとユーティリティが連携して動作する様子を示しています。

図 4-12. コード開発フロー



4.3.2 プロジェクトのビルド・オプション

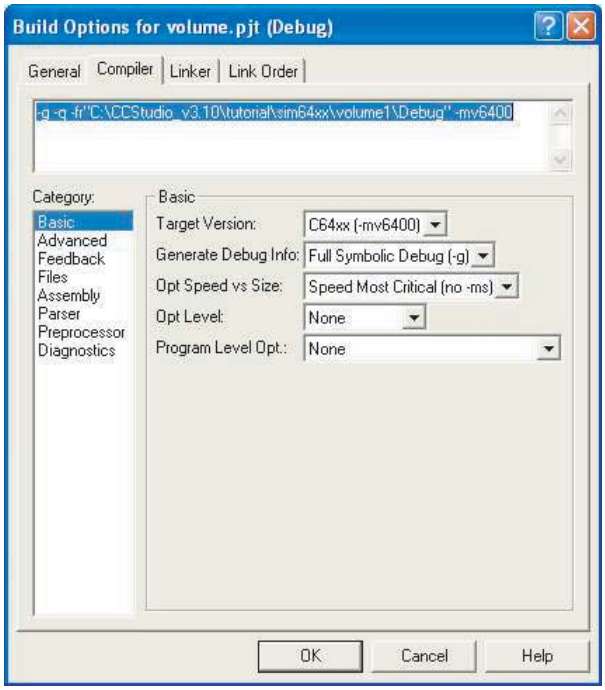
コード生成ツールを使用するためのグラフィカル・インターフェイスが用意されています。Code Composer Studio プロジェクトには、ターゲット・プログラムまたはライブラリをビルドするために必要なすべての情報が記録されています。プロジェクトに記録されている情報は、次のとおりです。

- ソース・コードおよびオブジェクト・ライブラリのファイル名
- コンパイラ、アセンブラ、およびリンカのオプション
- インクルード・ファイルの依存関係

プロジェクトをビルドする際に、CCStudio は適切なコード生成ツールを起動し、プログラムのコンパイル、アセンブル、およびまたは、リンクを行います。

コンパイラ、アセンブラ、およびリンカのオプションは、「Build Options」ダイアログ・ボックスで指定します（図 4-13 を参照）。このダイアログ・ボックスには、ほとんどすべてのコマンドライン・オプションが表示されます。表示されないオプションは、ダイアログ・ボックスの上部にある編集可能なテキスト・ボックス内に直接入力して指定できます。各ターゲット構成には、デバイス固有のオプション・セットがあります。詳細は、ターゲットのコンパイラまたはアセンブリ・マニュアルを参照してください。

図 4-13. 「Build Options」ダイアログ・ボックス



プログラムをビルドするときに使用されるコンパイラとリンカのオプションを設定できます。

ビルド・オプションは、使用頻度または設定内容に応じて、2 つの異なるレベルで設定できます。最初に、プロジェクト内のすべてのファイルに適用される一連のプロジェクト・レベル・オプションを指定できます。次に、ファイル固有のオプションを各ソース・コード・ファイルに指定して、プログラムを最適化することができます。

> **注：**
> 一般的に一緒に使用されるオプションについては、同じオプションを個別に繰り返し設定するのではなく、プロジェクト・レベルの構成に設定することをお勧めします。この情報については、オンライン・ヘルプおよびチュートリアルでも説明しています。

4.3.2.1 プロジェクト・レベルのビルド・オプションの設定

1. 「Project」→「Build Options」の順に選択します。
2. 「Build Options」ダイアログ・ボックスから、目的のタブを選択します。
3. プログラムをビルドするときに使用するオプションを選択します。
4. 「OK」をクリックして、設定内容を保存します。

4.3.2.2 ファイル固有のオプションの設定

1. 「Project　View」ウィンドウ内でソース・ファイルの名前を右クリックし、コンテキスト・メニューから「File Specific Options」を選択します。
2. このファイルをコンパイルするときに使用するオプションを選択します。これらのオプションは、プロジェクト・レベルのビルド・オプションとは異なります。
3. 「OK」をクリックして、設定内容を保存します。
4. 選択したファイルにのみ、変更内容が適用されます。

4.3.3 コンパイラの概要

C/C++ コンパイラは、標準 ISO/ANSI C プログラムをアセンブリ言語ソースに変換する、フル機能搭載の最適化コンパイラです。次の項では、これらのコンパイラの重要な機能について説明します。

4.3.3.1 Code Composer Studio IDE とのインターフェイス

コンパイラとのインターフェイスに関する機能は、次のとおりです。

- **コンパイラ・プログラム**
  コンパイラ・ツールには、1 つのステップでプログラムのコンパイル、アセンブリの最適化、アセンブル、およびリンクを行うために使用できるコンパイラ・プログラムが用意されています。詳細は、使用するデバイスの『最適化コンパイラ ユーザーズ・マニュアル』の「コンパイラ・プログラムについて」を参照してください。
- **フレキシブルなアセンブリ言語インターフェイス**
  コンパイラには単純化された呼び出し規約があるため、相互に呼び出し可能なアセンブリ関数と C 関数を記述することができます。詳細は、使用するデバイスの『最適化コンパイラ ユーザーズ・マニュアル』の「ランタイム環境」を参照してください。

4.3.4 アセンブリ言語開発ツール

アセンブリ言語開発ツールを次に示します。

- **アセンブラ**
  アセンブラは、アセンブリ言語のソース・ファイルを機械語のオブジェクト・ファイルに変換します。機械語は、共通オブジェクト・ファイル・フォーマット（COFF）に基づいています。
- **アーカイバ**
  アーカイバを使用すると、複数のファイルを「ライブラリ」と呼ばれる 1 つのアーカイブ・ファイルにまとめることができます。さらに、アーカイバを使用すると、メンバの削除、入れ替え、抽出、および追加を行って、ライブラリを変更できます。アーカイバを使用する主な目的の 1 つに、オブジェクト・モジュールのライブラリの構築があります。
- **リンカ**
  リンカは、複数のオブジェクト・ファイルを 1 つの実行可能なオブジェクト・モジュールに結合します。リンカが実行可能なモジュールを作成する際に、リンカは再配置を実行し、外部参照を解決します。リンカは、再配置可能な COFF オブジェクト・ファイルとオブジェクト・ライブラリを入力として受け入れます。
- **絶対リスタ**
  絶対リスタは、リンクされたオブジェクト・ファイルを入力として受け入れ、.abs ファイルを出力として作成します。これらの .abs ファイルをアセンブルして、相対アドレスではなく、絶対アドレスを含むリストを生成できます。絶対リスタを使用しない場合、このようなリストを生成するには、大変な労力が必要になります。
- **クロスリファレンス・リスタ**
  クロスリファレンス・リスタは、オブジェクト・ファイルを使用して、シンボル、シンボル

定義、リンクされたソース・ファイル内の参照を示すクロスリファレンス・リストを生成します。

- **Hex 変換ユーティリティ**
  Hex 変換ユーティリティは、COFF オブジェクト・ファイルを TI タグ付き、ASCII-Hex、インテル、Motorola-S、または Tektronix のいずれかのオブジェクト・フォーマットに変換します。変換されたファイルを EPROM プログラマにダウンロードできます。
- **ニーモニックから代数表記への変換ユーティリティ**
  このツールは、TMS320C54x デバイス専用の開発ツールで、アセンブリ言語のソース・ファイルを変換します。このユーティリティは、ニーモニック命令を含むアセンブリ言語のソース・ファイルを受け入れます。ニーモニック命令を代数表記命令に変換し、代数表記命令を含むアセンブリ言語のソース・ファイルを生成します。

4.3.5 アセンブラの概要

アセンブラは、アセンブリ言語のソース・ファイルを機械語オブジェクト・ファイルに変換します。これらのファイルは、共通オブジェクト・ファイル・フォーマット（COFF）になります。

2 パス・アセンブラは、次の処理を実行します。

- テキスト・ファイル内のソース文を処理して、再配置可能なオブジェクト・ファイルを作成する
- 要求があった場合には、ソース・リストを作成し、このリストを制御できるようにする
- コードをセクションに分割できるようにし、セクション・プログラム・カウンタ（SPC）をオブジェクト・コードのセクション毎に保持する
- グローバル・シンボルを定義および参照し、要求があった場合にはクロスリファレンス・リストをソース・リストに追加する
- 条件付きブロックをアセンブルする
- マクロをサポートして、マクロをインラインまたはライブラリ内で定義できるようにする

4.3.6 リンカの概要

リンカを使用すると、システム・メモリを構成する際に出力セクションを効果的にメモリ・マップに割り当てることができるようになります。リンカはオブジェクト・ファイルを結合する際に、次の作業を実行します。

- セクションをターゲット・システムの構成メモリに割り当てる
- シンボルとセクションを再配置し、最終アドレスに割り当てる
- 入力ファイル間の未定義の外部参照を解決する

リンカ・コマンド言語は、メモリ構成、出力セクションの定義、およびアドレスのバインディングを制御します。この言語では、式の割り当ておよび評価をサポートしています。設計するメモリ・モジュールを定義し、作成することでシステム・メモリを構成します。MEMORY と SECTIONS の 2 つの強力な擬似命令を使用すると、次の操作を行うことができます。

- セクションを特定のメモリ領域に割り当てる
- オブジェクト・ファイルのセクションを結合する
- リンク時にグローバル・シンボルを定義または再定義する

4.3.6.1 テキスト・ベースのリンカ

テキスト・リンカは、複数のオブジェクト・ファイルを 1 つの実行可能な COFF オブジェクト・モジュールに結合します。リンカ・コマンド・ファイル内のリンカ擬似命令を使用すると、オブジェクト・ファイルのセクションを結合し、セクションまたはシンボルをアドレスまたはメモリ範囲内にバインドし、グローバル・シンボルを定義または再定義できるようになります。詳細は、オンライン・ヘルプのトピック「Code Generation Tools」を参照してください

4.3.7 C/C++ 開発ツール

C/C++ 開発ツールに含まれるツールを次に示します。

- **C/C++ コンパイラ**
  C/C++ コンパイラは、C/C++ ソース・コードを受け入れ、アセンブリ言語のソース・コードを作成します。オプティマイザ、インターリスト・ユーティリティは、コンパイラの一部です。
  - コンパイラ・プログラムを使用すると、1 つのステップでソース・モジュールをコンパイル、アセンブル、およびリンクすることができます。いずれかの入力ファイルに、.sa 拡張子がある場合、コンパイラ・プログラムはアセンブリ・オプティマイザを起動します。
  - オプティマイザは、コードを変更して C プログラムの効率性を高めます。
  - インターリスト・ユーティリティは、C/C++ ソース文をアセンブリ言語の出力に差し込みます。
- **アセンブリ・オプティマイザ（C6000 のみ）**
  アセンブリ・オプティマイザを使用すると、パイプライン構造またはレジスタの割り当てを気にすることなく、直線的なアセンブリ・コードを記述することができます。レジスタが割り当てられていない、スケジュールされていないアセンブリ・コードを受け入れます。アセンブリ・オプティマイザは、レジスタを割り当て、ループ最適化を使用して、直線的なアセンブリ・コードをソフトウェア・パイプラインを利用する高度な並列的なアセンブリ・コードに変更します。
- **ライブラリ作成ユーティリティ**
  ライブラリ作成ユーティリティを使用すると、カスタマイズされたランタイム・サポート・ライブラリを構築することができます。標準のランタイム・サポート・ライブラリ関数は、rts.src 内のソース・コードにあります。ランタイム・サポート関数のオブジェクト・コードは、リトルエンディアン・モード対ビッグエンディアン・モード、および C コード対 C++ コードで、標準ライブラリにコンパイルされます。ランタイム・サポート・ライブラリには、コンパイラでサポートされる ISO 標準のランタイム・サポート関数、コンパイラ・ユーティリティ関数、浮動小数点算術演算関数、および C I/O 関数が含まれています。
- **C++ ネーム・デマングラ・ユーティリティ**
  C++ コンパイラは、関数の綴りをそのリンク・レベル名にエンコードすることにより、関数のオーバーロード、演算子のオーバーロード、およびタイプセーフなリンクを実現します。関数の綴りをリンク名にエンコードするプロセスは、ネーム・マングリングと呼ばれます。アセンブラ・ファイルまたはリンカ出力などのマングルされた名前を確認する場合、マングルされた名前と C++ ソース・コード内の対応する名前との関連付けが困難なことがあります。C++ ネーム・デマングラは、マングルされた名前を検出することにより、C++ ソース・コード内にあるオリジナルの名前に変換するデバッグ支援機能です。

4.4 Code Composer Studio プロジェクトのビルド

4.4.1 Code Composer Studio の使用

プログラムをビルドし、実行する手順は、次のとおりです。

1. 「Project」→「Rebuild All」の順に選択するか、「Rebuild All」ツールバー・ボタンをクリックします。プロジェクト内のすべてのファイルが、再コンパイル、再アセンブル、および再リンクされます。このプロセスのメッセージが、ウィンドウの下部にあるフレーム内に表示されます。
2. デフォルトでは、.out ファイルが現在のプロジェクト・フォルダの下にある debug ディレクトリにビルドされます。この場所を変更するには、「Select Configuration」ツールバーから他の場所を選択します。

3. 「File」→「Load Program」の順に選択します。再ビルドしたプログラムを選択し、「Open」をクリックします。このプログラムがターゲット DSP にロードされ、「Disassembly 」ウィンドウが開き、プログラムを構成する命令が逆アセンブルされて表示されます。
4. 「View」→「Mixed Source/ASM」の順に選択します。これにより、C ソースと結果のアセンブリ・コードを同時に表示することができます。
5. 混合モードのウィンドウ内でアセンブリ命令をクリックします （命令のアドレスや命令に渡されるフィールドではなく、実際の命令をクリックしてください)。
6. 「F1」キーを押します。Code Composer Studio IDE はその命令のヘルプを検索します。
7. 「Debug」→「Go Main」の順に選択して Main 関数から実行を開始します。実行が Main で停止します。
8. 「Debug」→「Run」の順に選択してプログラムを実行します。
9. 「Debug」→「Halt」の順に選択してプログラムの実行を停止します。

注：
CCStudio_v3.1¥cc¥bin ディレクトリ内にある提供されている timake.exe ユーティリティを使用すると、DOS シェルからプロジェクトをビルドできます。

4.4.2 外部メイク

Code Composer Studio では、外部メイクファイル（*.mak）と関連する外部 make ユーティリティを使用して、プロジェクト管理とビルド・プロセスのカスタマイズを行うことができます。

Code Composer Studio IDE でメイクファイルを使用してプログラムをビルドするには、そのメイクファイルを含む Code Composer Studio プロジェクトを作成する必要があります。Code Composer Studio プロジェクトをメイクファイルに関連付けると、プロジェクトとその内容が「Project View」ウィンドウ内に表示され、「Project」→「Build」コマンドと「Project」→「Rebuild All」コマンドを使用してプログラムをビルドできるようになります。

「Project View」ウィンドウ内でメイクファイルの名前をダブルクリックし、編集するメイクファイルを開きます。また、特別なダイアログ・ボックスを使用すると、メイクファイルのビルド・コマンドとオプションを変更することもできます。メイクファイルの操作では、通常の「Build Options」ダイアログ・ボックスは使用できません。複数の構成を作成し、それぞれに独自のビルド・コマンドとオプションを指定できます。

注：
制限と制約事項：「Project View」内でソース・ファイルをプロジェクトに追加したり、プロジェクトから削除することができます。ただし、「Project View」内で変更を加えても、メイクファイルの内容は変更されません。これらのソース・ファイルは、ビルド・プロセスに影響を与えたり、メイクファイルの内容にも反映されません。同様に、メイクファイルを編集しても、「Project View」内の内容が変更されることはありません。「Project View」内で追加されるソース・ファイルのファイル固有のオプションは、無効となります。「Project」→「Compile File」コマンドも使用できません。ただし、プロジェクトが保存される時に「Project View」の現在の状態は保持されます。

4.4.3 コマンドライン

4.4.3.1 コマンドラインからの timake ユーティリティの使用

CCStudio_v3.1¥cc¥bin ディレクトリ内にある timake.exe ユーティリティを使用すると、コマンド・プロンプトから、つまり Code Composer Studio の環境の外で、プロジェクト（*.prj）をビルドできます。このユーティリティを使用すると、バッチによるビルドが実行できます。

timake ユーティリティを起動する手順は、次のとおりです。

1. DOS コマンド・プロンプトを開きます。

2. DosRun.bat バッチ・ファイルを実行して事前に必要な環境変数をセットアップします。このバッチ・ファイルは、timake を使用する前に実行する必要があります。Code Composer Studio 製品を C:¥CCStudio_v3.1 にインストールした場合、このバッチ・ファイルは次の場所にあります。
   C:¥CCStudio_v3.1¥DosRun.bat
3. timake ユーティリティを実行します。

詳細は、オンラインヘルプのトピック「timake utility」を参照してください。

4.4.3.2 メイクファイル

Code Composer Studio IDE で外部メイクファイルを使用する方法以外に、標準の Code Composer Studio プロジェクト・ファイル（*.pjt）を標準のメイクファイルにエクスポートすることもできます。このメイクファイルを使用すると、標準のメイクファイルを使用してコマンドラインからビルドできます。Code Composer Studio には、標準の make ユーティリティ（gmake）が付属しており、DosRun.bat ファイルを実行後に実行できます。

Code Composer Studio プロジェクトを標準のメイクファイルにエクスポートする手順は、次のとおりです。

1. 「Project」ツールバー上の「Select Active Project」ドロップダウン・リストからプロジェクト名を選択して、目的のプロジェクトをアクティブにします。
2. 「Project」→「Export to Makefile」の順に選択します。
3. 「Exporting <filename>.pjt」ダイアログ・ボックスで、使用する make ユーティリティに対して、エクスポートする構成、デフォルト構成、ホストのオペレーティング・システムを指定します。また標準のメイクファイルに対するファイル名を指定します。
4. 「OK」をクリックして選択内容を受け入れ、標準のメイクファイルを生成します。

詳細は、オンライン・ヘルプのトピック「Exporting a Project to a Makefile」を参照してください。

4.5 利用可能な基盤ソフトウェア

4.5.1 DSP/BIOS

DSP/BIOS™ は、TMS320C5000™、TMS320C2000™、TMS320C6000™ の各 DSP プラットフォーム用に特別に設計された、スケーラブルなリアルタイム・カーネルです。DSP/BIOS を使用すると、従来の DSP ソフトウェア開発方法よりも迅速に高機能アプリケーションを開発し、運用することができます。また、独自のオペレーティング・システムまたは制御ループの開発および保守を行う必要がありません。マルチスレッドによりリアルタイム・アプリケーションがきちんと分割されるため、DSP/BIOS を使用するアプリケーションは保守しやすく、リアルタイム応答に影響を与えることなく新しい機能を追加できます。DSP/BIOS には、C2000、C5000 および C6000 の各 DSP プラットフォームで統一された API があるため、迅速なアプリケーションの移行が可能です。

アップデート・アドバイザを使用すると、DSP/BIOS API および構成ツールの更新されたバージョンをダウンロードすることができます。DSP/BIOS の更新されたバージョンをインストールした後に、コンポーネント・マネージャで CCStudio が使用する DSP/BIOS バージョンを指定することができます。

4.5.2 チップ・サポート・ライブラリ（CSL）

チップ・サポート・ライブラリ（CSL）には、オンチップ・ペリフェラルを構成および制御する C プログラム関数が用意されています。これは、実システムでアルゴリズムを実行するプロセスを単純化します。最終的な目的は、使いやすいペリフェラル、開発時間の短縮、移植性、ハードウェアの抽象化、デバイス間の小レベルの標準化と互換性です。

4.5.2.1 CSL の利点

CSL の利点を次に示します。

- **ペリフェラルをプログラムするための標準プロトコル**
  CSL には、オンチップ・ペリフェラル毎に高水準なプログラミング・インターフェイスが用意されています。CSL にはペリフェラルのレジスタ構成を定義するデータ・タイプとマクロがあり、各ペリフェラルのさまざまなオペレーションを実現する機能が組み込まれています。
- **基本的なリソース管理**
  多くのペリフェラルに対して open 関数と close 関数を使用する基本的なリソース管理が提供されています。これは、複数のチャネルをサポートするペリフェラルでは特に有効です。
- **シンボルを使用したペリフェラル記述**
  CSL 作成の副次的な利点として、すべてのペリフェラル・レジスタとレジスタ・フィールドの完全なシンボル記述が作成されることが挙げられます。最初の 2 項目で説明した高水準プロトコルを使用することをお勧めします。これら 2 つのプロトコルは、デバイス特定性が低く、コードをより新しいバージョンの DSP に移行しやすくなります。

4.5.3 ボード・サポート・ライブラリ（BSL）

TMS320C6000 DSK ボード・サポート・ライブラリ（BSL）は、すべてのオンボード・デバイスの構成および制御を行うために使用する C 言語アプリケーション・プログラミング・インターフェイス（API）です。これにより、開発者は実システム内で機能するアルゴリズムを入手することができます。BSL はビルドされ、ライブラリ・ファイルにアーカイブされる個別のモジュールで構成されています。各モジュールは個々の API を表し、API モジュールとして参照されます。モジュールが細分化されて構成されているので、LED と DIP の 2 つの API モジュールに分割される I/O ポート・モジュールを除き、各デバイスは単一の API モジュールでカバーされます。

4.5.3.1 BSL の利点

BSL の利点には、使いやすいデバイス、デバイス間の互換性レベル、開発時間の短縮、移植性、一部の標準化、ハードウェアの抽象化などがあります。

4.5.4 DSP ライブラリ（DSPLIB）

DSP ライブラリ（DSPLIB）には、C 呼び出し可能で、アセンブリが最適化された、汎用の信号処理およびイメージ / ビデオ処理用ルーチンが多く含まれています。通常これらのルーチンは、実行速度が最適であることが重要な、計算主体のリアルタイム・アプリケーションで使用されます。これらのルーチンを使用すると、標準の ANSI C 言語で記述された同等のコードよりも実行速度が著しく速くなります。また、DSP とイメージ / ビデオ処理関数が用意されているので、DSPLIB と IMGLIB を使用するとアプリケーションの開発時間を大幅に短縮できます。

DSPLIB の詳細は、次のマニュアルを参照してください。

- 『TMS320C54x DSP Library Programmer's Reference』(SPRU518)
- 『TMS320C55x DSP Library Programmer's Reference』(SPRU422)
- 『TMS320C62x DSP Library Programmer's Reference』(SPRU402)
- 『TMS320C64x DSP Library Programmer's Reference』(SPRU565)

4.5.4.1 DSPLIB の利点

DSPLIB には、一般的に使用されるルーチンがあります。用途に合わせて、関数を変更できるようにソース・コードが用意されています。

DSPLIB の特徴は、次のとおりです。

- 最適化されたアセンブリ・コード・ルーチン
- C およびリニア・アセンブリ・ソース・コード
- TI 最適化 C コンパイラと完全に互換性のある C 呼び出し可能なルーチン
- ベンチマーク（サイクルおよびコード・サイズ）
- 参照 C モデルに対してテスト済みであること

4.5.4.2 DSPLIB 関数の概要

DSPLIB は、C 呼び出し可能な高性能ルーチンを提供しています。これらのルーチンは、さまざまな信号処理およびイメージ / ビデオ処理アプリケーションの開発に対する主要な動機付けとなります。

DSPLIB 内に含まれているルーチンは、次の機能別カテゴリに構成されています。

- 適応フィルタ
- 相関
- FFT
- フィルタと畳み込み
- 数学演算
- マトリックス関数
- その他

4.5.5 イメージ / ビデオ処理ライブラリ（IMGLIB）

イメージ / ビデオ処理ライブラリ（IMGLIB）には、C 呼び出し可能で、アセンブリが最適化された、汎用の信号処理およびイメージ / ビデオ処理用ルーチンが多く含まれています。IMGLIB を使用できるのは、C5500/C6000 プラットフォーム・デバイスのみです。通常これらのルーチンは、実行速度が最適であることが重要な、計算主体のリアルタイム・アプリケーションで使用されます。これらのルーチンを使用すると、標準の ANSI C 言語で記述された同等のコードよりも実行速度が著しく速くなります。また、DSP とイメージ / ビデオ処理関数が用意されているので、DSPLIB と IMGLIB を使用するとアプリケーションの開発時間を大幅に短縮できます。

IMGLIB の詳細は、次のマニュアルを参照してください。

- 『TMS320C55x Image/Video Processing Library Programmer's Reference』(SPRU037)
- 『TMS320C62x Image/Video Processing Library Programmer's Reference』(SPRU400)
- 『TMS320C64x Image/Video Processing Library Programmer's Reference』(SPRU023)

4.5.5.1 IMGLIB の利点

IMGLIB には、一般的に使用されるルーチンがあります。用途に合わせて、関数を変更できるようにソース・コードが用意されています。

IMGLIB の特徴は、次のとおりです。

- 最適化されたアセンブリ・コード・ルーチン
- C およびリニア・アセンブリ・ソース・コード
- TI 最適化 C コンパイラと完全に互換性のある C 呼び出し可能なルーチン
- ベンチマーク（サイクルおよびコード・サイズ）
- 参照 C モデルに対してテスト済みであること

4.5.5.2 IMGLIB 関数の概要

IMGLIB は、C 呼び出し可能な高性能ルーチンを提供しています。これらのルーチンは、さまざまな信号処理およびイメージ / ビデオ処理アプリケーションの開発に対する主要な動機付けとなります。

IMGLIB 内に含まれているソフトウェア・ルーチンは、次の 3 つの異なる機能別カテゴリに分類されています。

- イメージ / ビデオの圧縮と解凍
- イメージ解析
- 画像フィルタ / フォーマット変換

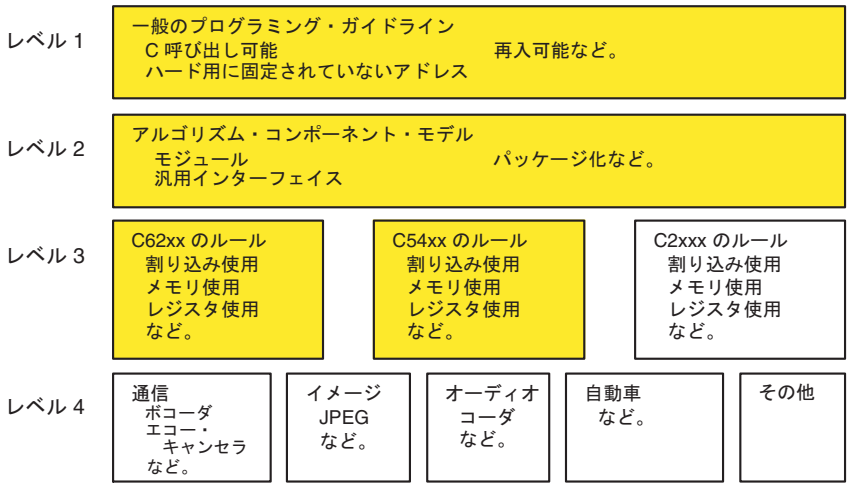
4.5.6 TMS320 DSP アルゴリズム・スタンダードの構成要素

DSP では、C 言語とアセンブリ言語を組み合わせてプログラムし、直接ハードウェア・ペリフェラルにアクセスします。性能上の理由により、標準のオペレーティング システムのサポートはほとんどまたは全くありません。汎用の組み込みマイクロプロセッサとは異なり、DSP は高度な信号処理アルゴリズムとヒューリスティックを実行するために設計されています。ただし、規格に整合性がないため、大幅にリエンジニアリングすることなく複数のシステムでアルゴリズムを使用することはできません。DSP アルゴリズムを再利用には、大きな労力を要するため、DSP ベースの新製品の製品化に要する時間は、月ではなく年単位で計測されます。

TMS320 DSP アルゴリズム・スタンダード（XDAIS）では、DSP アルゴリズムの一連の要件を規定しています。この要件を満たすと、システム・インテグレータは 1 つまたは複数のアルゴリズムを使用してシステムを迅速に組み立てることができます。

4.5.6.1 XDAIS の範囲

図 4-14. TMS320 DSP アルゴリズム・スタンダードの要素



レベル 1 には、アプリケーション・エリアに関係なく、すべての DSP アーキテクチャ上のすべてのアルゴリズムに適用されるプログラミング・ガイドラインが含まれています。最近開発されたソフトウェア・モジュールはほぼすべてこれらのガイドラインに従っているため、このレベルは形式化されたものです。

レベル 2 には、すべてのアルゴリズムを単一システムで動作させる規則とガイドラインが含まれています。データ・メモリのアルゴリズム使用と外部識別子の名前に関する規則が確立されているだけでなく、アルゴリズムをパッケージ化する方法に関する規則もあります。

レベル 3 には、特定の DSP ファミリーに関するガイドラインが含まれています。現在、アルゴリズムに対しては、プロセッサ・リソースの使用に関する合意に基づくガイドラインはありません。これらのガイドラインでは、各種アーキテクチャ上でのアルゴリズムの使用方法に関する概要が説明されています。これらのガイドラインから逸脱する可能性がある場合、アルゴリズム・ベンダはその逸脱内容を関連文書またはモジュール・ヘッダー内に明記してください。

図 4-14 内にある網掛けの枠は、このバージョンの仕様で扱っている分野を表しています。

レベル 4 には、さまざまなバーティカル・マーケットが含まれています。これらのビジネスはそれぞれ本質的に異なるため、マーケット・リーダーはバーティカル・マーケットに基づいたアルゴリズム群に対するインターフェイスを規定することが適切です。各々の独自のアルゴリズムをインターフェイスに指定しても、その規格は時流に追従できません。このレベルでは、上記 3 つのレベルで規定された規則に適合するアルゴリズムは、すべて eXpressDSP 準拠と見なされます。

4.5.6.2 規則およびガイドライン

TMS320 DSP アルゴリズム・スタンダードは、規則とガイドラインを指定しています。規則には、ソフトウェアを eXpressDSP 準拠にするために必ず従わなければなりません。一方、ガイドラインは、ソフトウェアを eXpressDSP 準拠にするために従うべき強く推奨された提案であり、必ず必要というわけではありません。

4.5.6.3 スタンダードの要件

XDAIS の必要な要素は、次のとおりです。

- 複数のベンダからのアルゴリズムを単一システムに統合可能。
- アルゴリズムはフレームワークにとらわれない。つまり、同じアルゴリズムをほとんどのアプリケーションまたはフレームワークでも効率的に使用できる。
- アルゴリズムを純粋に静的な環境だけでなく動的なランタイム環境でも運用可能。
- アルゴリズムをバイナリ形式で配布可能。
- アルゴリズムの統合にクライアント・アプリケーションの再コンパイルが不要。ただし、再構成と再リンクは必要な場合がある。

4.5.6.4 スタンダードの最終的な目的

XDAIS の最終的な目的は、次のとおりです。

- 開発者がスタンダードへ容易に適合できること
- 開発者がスタンダードへの適合を検証可能であること
- システム・インテグレータによる TI DSP 間の移行が簡単にできること
- 構成、性能のモデリング、スタンダードへの適合、デバッグなどシステム・インテグレータの作業を単純化できること
- 静的システムへ経費削減できること

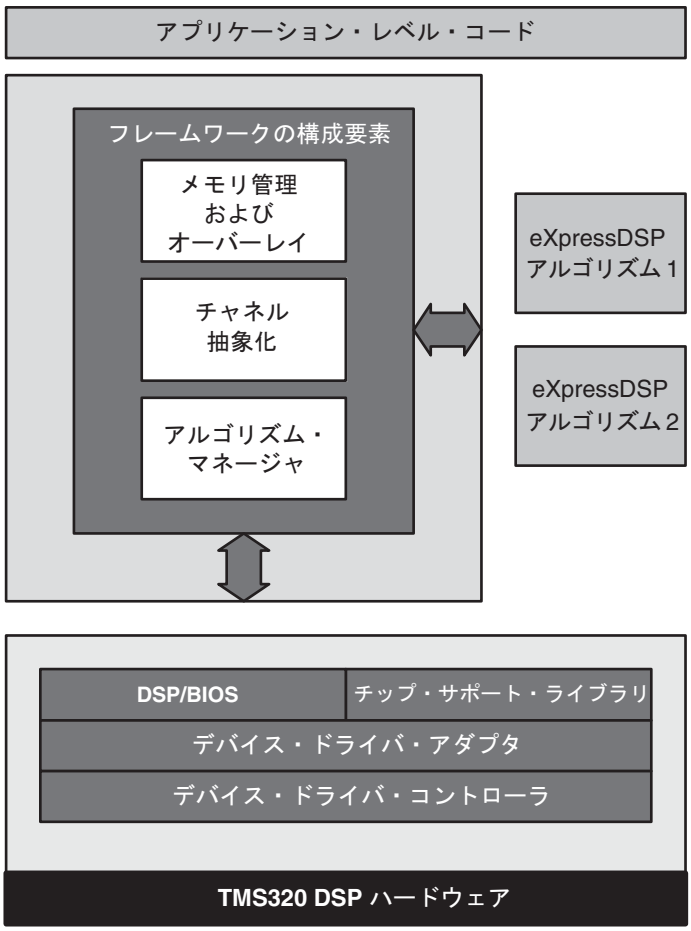
4.5.7 リファレンス・フレームワーク

eXpressDSP ソフトウェアのリファレンス・フレームワークが、DSP/BIOS と TMS320 DSP アルゴリズム・スタンダードを使用するアプリケーション用に用意されています。開発者は初めにシステムと将来のニーズに最も適合するリファレンス・フレームワークを選択し、そのフレームワークを改良し、eXpressDSP 準拠アルゴリズムに適合させます。デバイス・ドライバ、メモリ管理、およびチャネルのカプセル化などの一般的な要素は、フレームワーク内ですでに事前に構成されているため、開発者はシステムに集中して取り組むことができます。リファレンス・フレームワークには、TMS320C5000 および TMS320C6000 の各 DSP 用の設計済みで再利用可能な C 言語のソース・コードが組み込まれています。

リファレンス・フレームワークのソフトウェアと文書は、TI の Web サイトからダウンロードできます。Code Composer Studio のインストールには、含まれていません。

図 4-15 では、ターゲット DSP 上でリファレンス・フレームワークを構成する要素を示します。

図 4-15. リファレンス・フレームワークの要素



リファレンス・フレームワークの要素を順に説明します。

- **デバイス・コントローラとデバイス・アダプタ**
  リファレンス・フレームワークで使用されるデバイス・ドライバは、デバイス・アダプタを備え、標準のデバイス・コントローラ・インターフェイスを指定する標準のドライバ・モデルに基づいています。独自の外部ハードウェアがある場合、デバイス・コントローラを変更する必要がありますが、デバイス・アダプタはほとんど、または全く変更する必要がありません。
- **チップ・サポート・ライブラリ（CSL）**
  デバイス・コントローラは、チップ・サポート・ライブラリ・モジュールを使用してペリフェラル・ハードウェアをサポートします。
- **DSP/BIOS**
  この拡張可能なソフトウェア・カーネルは、各リファレンス・フレームワークがどのようにさまざまな eXpressDSP 基盤を必要に応じて利用するかを示す良い例です。ローエンドの RF1 フレームワークは、比較的少ない DSP/BIOS モジュールを使用します。占有スペースの節約とともに、モジュール数の削減により、多数のモジュールから生じる問題を望まない設計者は設計上の選択を明確にすることができるようになります。
- **フレームワークの構成要素**
  これらの要素は、全体のシステム・リソース管理を行うように、設計されています。この例

の 1 つに、チャネルの抽象化があります。すべてのリファレンス・フレームワークには、何らかのチャネル管理が必要です。ただし、使用される可能性のあるチャネル数に基づいて、設計の最適化を行うことができます。チャネル数が 1 ～ 3 の場合、チャネルのスケジュールはオーバーヘッドの少ない DSP/BIOS の HWI モジュールと IDL モジュールで処理されます。チャネル数が多数ある場合、SWI モジュールを使用する方が賢明です。ただし、占有スペースがさらに必要になります。動的に変更されるチャネルがある大規模システムの場合、TSK モジュールが最適です。チャネル・マネージャと同様、アルゴリズム・マネージャは一部の eXpressDSP 準拠アルゴリズムを管理します。その他のフレームワーク構成要素は、メモリ・オーバーレイ方式を処理するモジュールです。これは、メモリに制約のある、ほとんどのシステムで重要な技術です。適切なフレームワークから開始することにより、開発に関わる多数の選択項目が単純化されます。

- **eXpressDSP 準拠アルゴリズム**
  各アルゴリズムは、『TMS320 DSP Algorithm Standard Rules and Guidelines』(SPRU352) に記述されている規則とガイドラインに準拠しています。スタンダードに準拠するには、アルゴリズムはハードウェア・ペリフェラルに直接アクセスしてはいけません。また、IALG（メモリ管理用）とオプションの IDMA（DMA リソース管理用）と呼ばれる標準のリソース管理インターフェイスを実装する必要があります。TI では、アルゴリズムは有限インパルス応答（FIR）フィルタやボリューム・コントローラなど単純な例を提供しています。TI が提供するアルゴリズムの代わりに、より有効な eXpressDSP 準拠アルゴリズムを使用することもできます。つまり、汎用リファレンス・フレームワークをよりアプリケーション固有なものにすることです。
- **アプリケーション・レベル・コード**
  最後のステップは、アプリケーション・レベル・コードを変更することです。このコードを使用することにより、独自の付加価値をもつアプリケーション固有の知識が適用され、本当の意味での製品の差別化が可能になります。たとえば、単一チャネルの MP3 プレーヤに必要なアプリケーション・コードとデジタル補聴器に必要なアプリケーション・コードは、異なります。

4.6 自動化（プロジェクト管理）

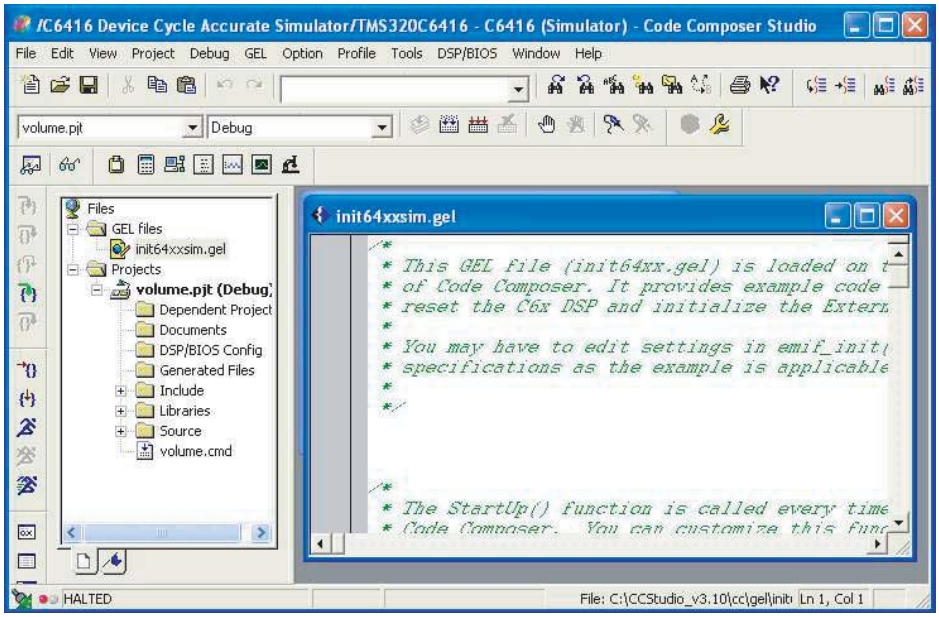
4.6.1 GEL（General Extension Language）の使用

GEL（General Extension Language）は、C 言語によく似たインタープリタ型言語です。Code Composer Studio の利便性を高める関数を作成できます。GEL 関数を作成するには、GEL 文法を使用し、その関数を Code Composer Studio IDE にロードします。GEL 関数のサブセットを使用してプロジェクトのビルド・オプションを自動化したり、カスタム GEL メニューを作成してプロジェクトを自動的に開いてビルドしたりすることができます。

volume プロジェクトをオープンするための GEL スクリプトの例を示します。

```
/*
 * Copyright 1998 by Texas Instruments Incorporated.
 * All rights reserved. Property of Texas Instruments Incorporated.
 * Restricted rights to use, duplicate or disclose thiscode are
 * granted through contract.
*/
/*
 * ======== PrjOpen.gel ========
 * Simple gel file to demonstrate project management
capabilities of GEL
*/menuitem "MyProjects"
hotmenu OpenVolume()
{
// Open Volume tutorial example
GEL_ProjectLoad("C:\\CCStudio_v3.1\\tutorial\\sim55xx\\volume1\\volume.pjt");
// Set currently active configuration to debug
GEL_ProjectSetActiveConfig("C:\\CCStudio_v3.1\\tutorial\\sim55xx\\volume1\\volume.pjt",
"Debug");
// Build the project.
GEL_ProjectBuild();
}
```

図 4-16. カスタム GEL ファイル



4.6.2 スクリプト・ユーティリティ

スクリプト・ユーティリティ（Scripting Utility）は、VB や Perl のスクリプト言語に組み込まれる IDE のコマンド・セットです。Perl や VB などのスクリプト言語の機能を活用し、Code Composer Studio 内で実行される自動化タスクと組み合わせることができます。スクリプト・ユーティリティを使用すると、テスト・シナリオを構成したり、対応プロジェクトを開いてビルドしたり、実行用にプロジェクトをロードしたりできます。プロジェクトのビルドと管理に使用できるスクリプト・コマンドが多数用意されています。スクリプトは、同期化を行います。

スクリプト・ユーティリティは、アップデート・アドバイザを通して入手できるアドオン機能です（7.2 節を参照）。

第 5 章

デバッグ

この章の説明は、Code Composer Studio IDE を使用するすべてのプラットフォームに適用されます。ただし、すべてのデバイスがこの章で説明するすべてのツールにアクセスできるわけではありません。使用可能なツールについては、Code Composer Studio IDE に付属するオンライン・ヘルプとオンライン文書を参照してください。

この章では、Code Composer Studio に付属する各種デバッグ・ツールについて説明します。

項目 **ページ**

5.1 デバッグ用の環境のセットアップ

アプリケーションを適切にデバッグするには、事前に環境を設定しておく必要があります。その設定方法を以降に示します。

5.1.1 カスタム・デバッグ・オプションの設定

一部のデバッグ・オプションは、Code Composer Studio IDE でカスタマイズ可能です。デバッグをし易くさせたり、好みに合うようにするために、これらのオプションを構成することができます。

5.1.1.1 「Debug Properties」タブ

この「Debug Properties」タブは、「Option」→「Customize」の順に選択して表示される「Customize」ダイアログ・ボックスにあります。デバッグ時に特定のデフォルト動作をオフにすることができます。他のオプションは、オンライン・ヘルプを参照してください。「Debug Properties」タブで設定できる動作は、次のとおりです。

- **Open the Disassembly Window automatically**
  このオプションを無効にすると、プログラムがロードされたときに「Disassembly」ウィンドウが表示されません。このオプションは、デフォルトで有効になっています。
- **Perform Go Main automatically**
  このオプションを有効にすると、自動的にロードされるアプリケーションのシンボル「main」にデバッガが移動します。このオプションは、デフォルトで無効になっています。
- **Connect to the target when a control window is opened**
  コントロール・ウィンドウとは、Code Composer Studio の IDE インターフェイス全体を表します。PDM 実行時にこのインスタンスを複数開くことができます。ターゲット接続に問題がある場合、または実際のターゲットに接続する必要がない場合（ソース・コード記述時など）、このオプションを無効にすることができます。このオプションは、デフォルトで無効になっています。
- **Remove remaining debug state at connect**
  Code Composer Studio IDE はターゲットから切断するときに、通常デフォルトでブレークポイントの解除を行います。何らかの理由でブレークポイントを解除する際にエラーが発生した場合、Code Composer Studio はターゲットに再接続する際に再度ブレークポイントの解除を行います。ただし、このブレークポイントを解除する 2 回目の動作により、ターゲットの状態が悪くなることがあります。このため、再接続時にはブレークポイントを解除する 2 回目の動作を無効にしてください。
- **Animation speed**
  Animation speed とは、ブレークポイント間の最小時間（秒数）を表します。前のブレークポイントからこの最小時間が経過しないと、プログラム実行は再開されません。詳細は、5.2.1.1 項を参照してください。

5.1.1.2 「Directories」タブ

アプリケーションが停止した時点でソース・ファイルを開くには、デバッガはその場所を必要とします。デバッガには、開いているプロジェクト内のすべてのファイル、現在のディレクトリ内のファイル、および Code Composer Studio IDE に対して指定したパスのソース・パス情報が含まれています。

ユーザーが Code Composer Studio IDE に対して指定するパスは、デフォルトでは何もありません。このため、デバッグ中のアプリケーションは、開いているプロジェクトまたは現在のディレクトリ内に存在しないソース・ファイルを使用している場合、これらのファイルに対するパスを指定する必要があります。指定しないと、ソース・ファイルを参照する場所で、プログラムの実行が停止したときに、デバッガは自動的にそのファイルを開くことができません。この場合、ユーザーに対してファイルを指定するようにプロンプトが表示されます。たとえば、開いたプロジェクト

内にソース・ファイルがない例としては、ライブラリがビルドに組み込まれている場合が一般的です。

「Directories」タブでは、含まれているソース・ファイルを検索するためにデバッガが使用する検索パスを追加指定できます。

検索パス・ディレクトリを指定するには、「Option」→「Customize」の順に選択したときに表示されるダイアログ・ボックスから「Directories」タブを選択します。ダイアログ・ボックスの上部にあるスクロール矢印を使用してタブを選択する必要があります。ここでオプションは、次のとおりです。

- **Search the following directories for source files during debugging**
  ここには、指定した検索パスのディレクトリ・リストが表示されます。デバッガは、これらのディレクトリを上から下の順に検索します。2 つのファイルが同じ名前で、別々のディレクトリにある場合、ディレクトリ・リスト内の一番上に表示されているディレクトリ内にあるファイルが優先されます。
- **New**
  新しいディレクトリをディレクトリ・リストに追加するには、「New」をクリックします。フルパスを入力するか、適切なディレクトリを指定します。デフォルトでは、新しいディレクトリがリストの一番下に追加されます。
- **Delete**
  ディレクトリをリストから削除するには、ディレクトリ・リストからディレクトリを選択し、「Delete」をクリックします。
- **Move Up/Move Down**
  リスト内の指定したディレクトリを上方向に移動するには、ディレクトリ・リストからディレクトリを選択してから、「Move Up」をクリックします。リスト内の指定したディレクトリを下方向に移動するには、ディレクトリ・リストからディレクトリを選択してから、「Move Down」をクリックします。
- **Look in subfolders**
  デバッガが、リスト表示されたパスのサブフォルダ内を検索できるようにします。
- **Default File I/O Directory**
  ソース・ファイルのディレクトリの設定に加え、「Default File I/O Directory」オプションを有効にすると、File I/O ファイルのデフォルト・ディレクトリをセットすることができます。参照ボタンを使用して、デフォルト・ディレクトリとして選択するパスを指定します。

5.1.1.3 「Program Load Options」タブ

「Option」→「Customize」の順に選択したときに表示されるダイアログ・ボックスから「Program/Project Load」タブを選択すると、プログラムのロードに関するデフォルトを設定することができます。

- **Perform verification during Program Load**
  デフォルトでは、このチェックボックスにチェックマークが付いています。これにより、プログラムが正しくロードされたかどうかが選択メモリの読み取りにより検証されます。
- **Load Program After Build**
  このオプションが選択されると、プロジェクトがビルドされた直後に実行可能なプログラムがロードされます。これにより、ターゲットにはビルド後に生成された最新のシンボル情報が組み込まることになります。
- **Do Not Set CIO Breakpoint At Load**
  デフォルトでは、TI ランタイム・ライブラリ（rts*.lib）を使用してプログラムがリンクされている場合、プログラムのロード時に C I/O ブレークポイント（C$$IO$$）が設定されます。このオプションを有効にすると、C I/O ブレークポイントは設定されません。C I/O ブレークポイントは、printf や scanf などの C I/O ライブラリ関数の通常動作に必要です。プログラムが

CIO 関数を実行しない場合、C I/O ブレークポイントは必要ありません。C I/O コードが RAM にロードされると、ソフトウェア・ブレークポイントが設定されます。ただし、C I/O コードが ROM にロードされると、ハードウェア・ブレークポイントが使用されます。ほとんどのプロセッサでは少数のハードウェア・ブレークポイントしかサポートされていないため、1 つ使用するだけでデバッグ時に大きな影響があります。C I/O コードが ROM にロードされる際にハードウェア・ブレークポイントを使用しないようにするには、ブレークポイントをコード内に組み込み、このブレークポイントが組み込みのブレークポイントであることを示すように名前を C$$IO$$ から C$$IOE$$ に変更します。

- **Do Not Set End of Program Breakpoint At Load**
  デフォルトでは、TI ランタイム・ライブラリ（rts*.lib）を使用してプログラムがリンクされている場合、プログラムのロード時に End of Program ブレークポイント（C$$EXIT）が設定されます。このオプションを有効にすると、End of Program ブレークポイントは設定されません。End of Program ブレークポイントは、プログラムが実行を終了する前にプロセッサを停止させるために使用します。プログラムが無限ループを実行する場合は、End of Program ブレークポイントは必要ありません。End of Program コードが RAM にロードされると、ソフトウェア・ブレークポイントが設定されます。ただし、End of Program コードが ROM にロードされると、ハードウェア・ブレークポイントが使用されます。ほとんどのプロセッサでは少数のハードウェア・ブレークポイントしかサポートされていないため、1 つ使用するだけでデバッグ時に大きな影響があります。End of Program コードが ROM にロードされる際にハードウェア・ブレークポイントを使用しないようにするには、ブレークポイントをコード内に組み込み、このブレークポイントが組み込みのブレークポイントであることを示すように名前を C$$EXIT から C$$EXITE$$ に変更します。
- **Disable All Breakpoints When Loading New Programs**
  このオプションを有効にすると、新規プログラムをロードする前にすべての既存ブレークポイントが解除されます。
- **Open Dependent Projects When Loading Projects**
  デフォルトでは、プログラムにメイン・プロジェクトが依存するサブプロジェクトがある場合、メイン・プロジェクトと同時にすべてのサブプロジェクトが開きます。このオプションが無効の場合、サブプロジェクトは開きません。
- **Do Not Scan Dependencies When Loading Projects**
  インクリメンタル・ビルド中にコンパイルする必要があるファイルを判別するために、プロジェクトは各ソース・ファイルのインクルード・ファイルの依存関係を保持しなければなりません。プロジェクトをビルドすると、常に依存関係のツリーが作成されます。依存関係ツリーを作成するために、プロジェクト・リスト内のすべてのソース・ファイルから #include、.include、.copy の各擬似命令が再帰的にスキャンされ、各ファイル名がプロジェクト・リストに追加されます。デフォルトでは、プロジェクトを開くと、プロジェクト内のすべてのファイルの依存関係がスキャンされます。このオプションが無効になっていると、プロジェクトを開くときに依存関係を自動的にスキャンしません。これにより、プロジェクトを開く時間が速くなります。

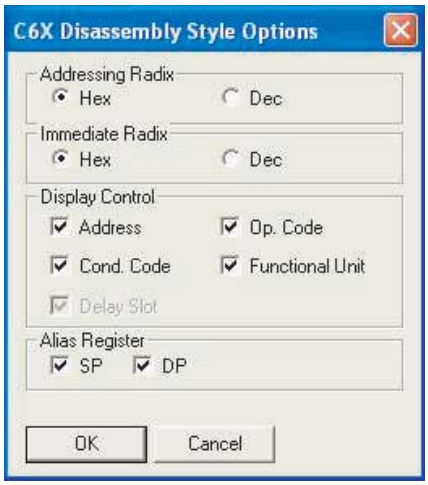
5.1.1.4 Disassembly Style の設定

「Disassembly」ウィンドウに情報を表示する方法を変更するオプションが用意されています。「Disassembly Style Options」ダイアログ・ボックスを使用すると、デバッグ・セッション用の表示オプションを入力できます。

Disassembly Style Options を設定する手順は、次のとおりです。

1. 「Option」→「Disassembly Style」の順に選択するか、「Disassembly」ウィンドウ内で右クリックして「Properties」→「Disassembly Options」の順に選択します。
2. 「Disassembly Style Options」ダイアログ・ボックス内で好みに合わせて入力します。

図 5-1. 「Disassembly Style Options」ダイアログ・ボックス



3. 「OK」をクリックします。「Disassembly」ウィンドウの内容が、新しいスタイルでただちに更新されます。

5.1.2 シミュレーション

シミュレータを設定して、実際のハードウェア・ターゲットのような動作をさせるには、メモリ・マッピング（5.1.3 項を参照）、ピン接続（5.1.4 項を参照）、ポート接続（5.1.5 項を参照）の各オプションを設定します。

5.1.3 メモリ・マッピング

メモリ・マップは、アクセス可能なメモリ領域およびアクセスできないメモリ領域をデバッガに指示します。メモリ・マップは、アプリケーションによって異なります。

メモリ・マップが定義され、メモリ・マッピングが有効になると、デバッガはメモリ・マップに応じてすべてのメモリ・アクセスをチェックします。デバッガは、メモリ・マップにより保護されているメモリ領域にはアクセスしません。

デバッガはメモリ・アクセスとメモリ・マップを比較する際に、ハードウェアではなくソフトウェア内で実行します。プログラムが存在しないメモリにアクセスしようとしてもデバッガでは防止できません。

5.1.3.1 シミュレーション用のメモリ・マッピング

シミュレータは、シミュレートする DSP ターゲットに対する有効なメモリ設定を最も一般的に表現するために事前に定義されたメモリ・マップ範囲を利用します。メモリ・マップの設定はある程度まで変更できますが、有効なメモリ範囲に大幅な変更を加えるとシミュレータのパフォーマンスに影響を与える場合があるため、このような変更はお勧めしません。

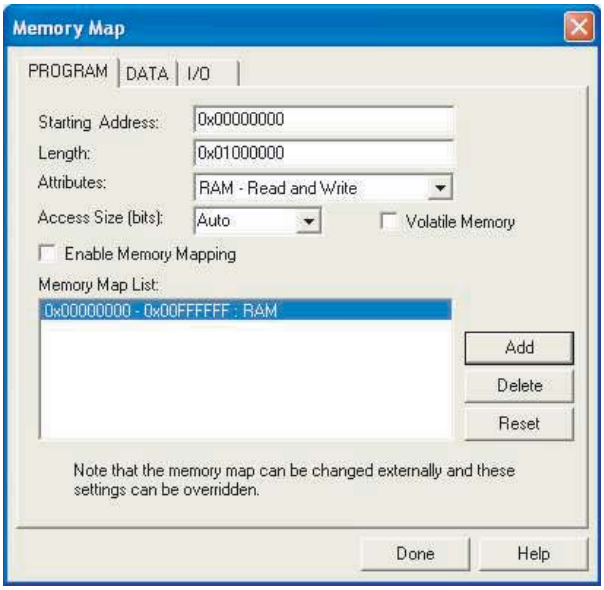
5.1.3.2 デバッガ用のメモリ・マッピング

メモリ・マップは、デバッガの使用中に対話形式で定義することができますが、デバッグを開始するときは事前に 1 つのメモリ・マップを設定してから、このメモリ・マップをその他すべてのデバッグ・セッションに使用するため、これは不便なものになります。

メモリ・マップ範囲を新規に追加する手順は、次のとおりです。

1. 「Option」→「Memory Map」の順に選択します。

図 5-2. メモリ・マップ



2. 実際の、またはシミュレートするターゲット・メモリ構成で複数のページがサポートされている場合、「Memory Map」ダイアログ・ボックスにはメモリ・ページのタイプ毎に別々のタブ（Program、Data、I/O など）があります。変更するメモリのタイプ用のタブを選択します。1 つのメモリ・ページしかないプロセッサの場合、複数のタブは表示されません。「Memory Map」ダイアログ・ボックスのオプションは、次のとおりです。

   - **Enable Memory Mapping**

     チェックボックスにチェックマークが付いていることを確認します。チェックマークが付いていないと、ターゲット上のすべてのアドレス指定可能メモリ（RAM）が有効であるとデバッガは想定します。

   - **Starting Address**

     入力フィールドに新規メモリ範囲の開始アドレスを入力します。

   - **Length**

     入力フィールドに新規メモリ範囲の長さを入力します。

   - **Attributes**

     新規メモリ範囲の read/write 特性を選択します。

   - **Access Size (bits)**

     ターゲット・プロセッサのアクセス・サイズを指定します。ドロップダウン・リストからアクセス・サイズを選択するか、フィールドに数値を入力します。1 つのアクセス・サイズしかサポートしていないプロセッサの場合、サイズを指定する必要はありません。

   - **Volatile Memory**

     通常、書き込みアクセスは、Read、Modify、Write のそれぞれの動作で構成されています。「Volatile Memory」オプションがメモリのセグメントで設定されている場合、そのメモリへの書き込みアクセスはすべて Write 動作のみを使用して実行されます。

   - **Memory Map List**

     メモリ・マップ範囲のリストが表示されます。

   - **Add**

     新規メモリ範囲を「Memory Map List」に追加します。

- **Delete**
  「Memory Map List」で、削除するメモリ・マップ範囲を選択し、「Delete」ボタンをクリックします。また、「Attributes」フィールドを「None - No Memory/Protected」に変更して、既存のメモリ・マップ範囲を削除することもできます。これは、このメモリ・ロケーションには読み取りも書き込みもできないことを意味します。
- **Reset**
  「Memory Map List」に、デフォルト値をセットします。

3. 「Done」をクリックして、設定内容を保存します。

デバッガは、既存のメモリ範囲と重複する新規メモリ範囲の入力を認めています。新規範囲は有効と見なされ、重複した範囲の属性はそれに応じて変更されます。

メモリ・マップを定義した後に、その read/write 属性を変更したい場合があります。属性を変更するには、同じ開始アドレスと長さをもつ新規メモリ・マップを定義し、「Add」ボタンをクリックします。既存の属性が新しい属性に上書きされます。

5.1.3.3 GEL によるメモリ・マップの定義

メモリ・マップは、GEL（General Extension Language）組み込み関数を使用しても定義することができます。GEL には、完全なメモリ・マッピング関数のセットがあります。メモリ・マップを実装する最も簡単な方法は、メモリ・マッピング関数を GEL テキスト・ファイル内に置き、その GEL ファイルをスタートアップ時に実行することです（GEL の概要については、5.1.5 項を参照）。

Code Composer Studio IDE をはじめて起動したとき、メモリ・マップはオフになっています。任意のメモリ・ロケーションにアクセスでき、メモリ・マップによる干渉はありません。オプションの GEL ファイル名をパラメータとして指定して Code Composer Studio を起動すると、その GEL ファイルが自動的にロードされます。このファイルに GEL 関数 StartUp() が含まれている場合、ファイル内の GEL 関数が実行されます。このファイル内で GEL マッピング関数を指定することで、使用する環境に合わせたメモリ・マッピング要件を自動的に定義することが可能になります。

次の GEL 関数を使用して、メモリ・マップを定義します。

表 5-1. メモリ・マップの GEL 関数

関数	説明
GEL_MapAdd()	メモリ・マップの追加
GEL_MapDelete()	メモリ・マップの削除
GEL_MapOn()	メモリ・マップを有効にする
GEL_MapOff()	メモリ・マップを無効にする
GEL_MapReset()	メモリ・マップをリセットする

GEL_MapAdd() 関数は、有効なメモリ範囲を定義し、メモリ範囲の read/write 特性を識別します。次に示す GEL ファイルのサンプルは、読み取りも書き込みも可能な長さ 0xF000 の 2 つのブロックを定義するのに使用できます。

```
StartUp()
{
GEL_MapOn();
GEL_MapReset();
GEL_MapAdd(0, 0, 0xF000, 1, 1);
GEL_MapAdd(0, 1, 0xF000, 1, 1);
}
```

メモリ・マップを設定したら、「Option」→「Memory Map」の順に選択して、メモリ・マップを表示します。

5.1.4 ピン接続

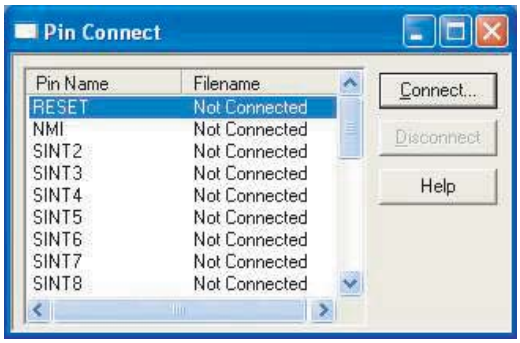
ピン接続ツールを使用すると、選択されている外部割り込みが発生する間隔を指定できます。

外部割り込みをシミュレートする手順は、次のとおりです。

1. 割り込み間隔を指定するデータ・ファイルを作成します。
2. 「Tools」メニューから「Pin Connect」を選択して、ピン接続ツールを起動します。
3. ピン名を選択し、「Connect」をクリックします。
4. プログラムをロードします。
5. プログラムを実行します。

ピン接続ツールの詳細は、オンライン・ヘルプのピン接続に関するトピックを参照してください。「Help」→「Contents」→「Debugging Window & Analysis Tools」→「Analysis Tools (Generic)」→「Pin Connect」の順に選択します。

図 5-3. ピン接続ツール



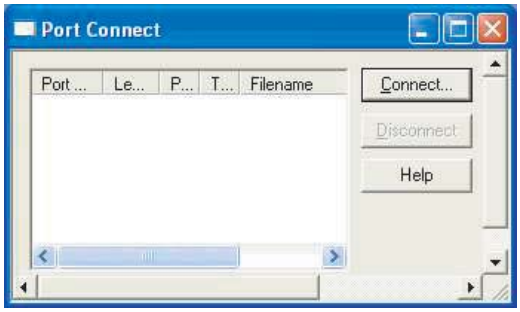
5.1.5 ポート接続

ポート接続ツールを使用すると、メモリ・アドレスを介してファイルにアクセスできます。メモリ（ポート）アドレスに接続することにより、ファイルからデータを読み込んだり、ファイルへデータを書き出したりできるようになります。

メモリ（ポート）アドレスをデータ・ファイルに接続する手順は、次のとおりです。

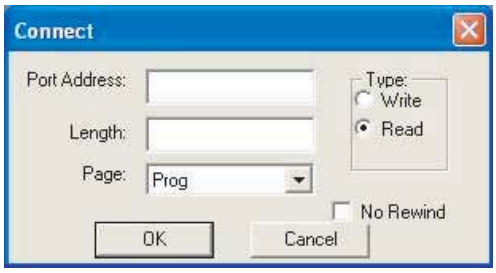
1. 「Tools」メニューから、「Port Connect」を選択します。「Port Connect」ウィンドウが表示され、ポート接続ツールが起動されます。

図 5-4. ポート接続ツール



2. 「Connect」ボタンをクリックすると「Connect」ダイアログ・ボックスが開きます。

図 5-5. ポート・アドレス接続



3. 「Port Address」フィールドに、メモリ・アドレスを入力します。このパラメータには、絶対アドレス、任意の C 式、C 関数の名前、またはアセンブリ言語のラベルを指定できます。16 進数のアドレスを指定する場合、必ずアドレス番号の先頭に 0x を指定してください。指定しないと、10 進数のアドレスとして処理されます。
4. 「Length」フィールドに、メモリ範囲の長さを入力します。長さには、任意の C 式を指定できます。
5. 「Page」フィールド（C5000 のみ）で、アドレスが占有するメモリのタイプ（プログラムまたは I/O）を選択します。プログラム・メモリの場合、「Prog」を選択します。I/O 空間の場合、「I/O」を選択します。
6. 「Type」フィールドで、ファイルにデータを書き込むか、ファイルからデータを読み取るかに応じて、Write（書き込み）または Read（読み取り）のオプション・ボタンを選択します。
7. 「OK」をクリックすると、「Open Port File」ウィンドウが表示されます。
8. 接続するデータ・ファイルを選択し、「Open」をクリックします。
9. 「No Rewind」機能を選択して、ファイルの終わり（EOF）に到達した際にファイルの先頭に移動しないようにします。EOF 後の読み取りアクセスでは、0xFFFFFFFF の値が読み取られ、ファイルのポインタは変更されません。

このファイルは、関連付けられているメモリ・アドレスに対するアセンブリ言語の read または write 命令中にアクセスされます。任意のメモリ・アドレスをファイルに接続可能です。1 つのメモリ・アドレスに、1 つの入力ファイルと 1 つの出力ファイルを接続できます。複数のアドレスを 1 つのファイルに接続することができます。

ポート接続ツールの詳細は、オンライン・ヘルプのポート接続に関するトピックを参照してください。「Help」→「Contents」→「Debugging Window & Analysis Tools」→「Analysis Tools (Generic)」→「Port Connect」の順に選択します。

5.1.6 プログラムのロード

プログラムをビルドすると作成される COFF ファイル（*.out）を実行する前に、実際の、またはシミュレートするターゲット・ボードにロードする必要があります。

プログラム・コードとデータは、ターゲット上の COFF ファイルに指定されたアドレスにダウンロードされます。シンボルは、ホスト上のデバッガに保持されているシンボル・テーブルにロードされます。これらのシンボルは、COFF ファイルに指定されたコードとデータの各アドレスにロードされます。

COFF ファイルをロードするには、「File」→「Load Program」の順に選択し、「Load Program」ダイアログ・ボックスから目的の COFF ファイルを選択します。

5.1.6.1 シンボル情報のみのロード

デバッガがオブジェクト・コードをロードできないか、またはロードが不要なデバッグ環境で作業する場合（コードが ROM 内にある場合など）は、シンボル情報のみをロードすると便利です。

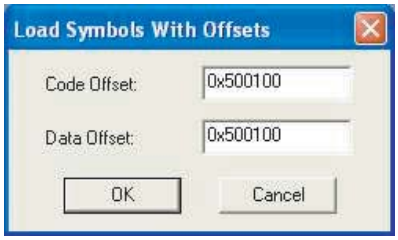
シンボルをロードするには、メイン・メニューから「File」→「Load Symbols」→「Load Symbols Only」の順に選択し、「Load Symbols」ダイアログ・ボックスから目的の COFF ファイルを選択します。

デバッガは、ホストに保持されているシンボル・テーブルから事前にロードされたシンボルを削除します。その後、シンボル・ファイル内のシンボルがシンボル・テーブルにロードされます。シンボルは、シンボル・ファイルに指定されたコードとデータの各アドレスにロードされます。このコマンドは、メモリを変更したり、プログラムのエントリ・ポイントを設定したりしません。

デバッガが指定されたシンボル・ファイル内のすべてのシンボルに適用する、コード・オフセットとデータ・オフセットを指定することもできます。たとえば、0x100 で始まるコード・アドレスと 0x1000 で始まるデータ・アドレスを含む、実行可能なプログラムのシンボル・ファイルがあると仮定します。ただし、ターゲット上にロードされるプログラムでは、対応するコードは 0x500100 で始まり、データは 0x501000 に配置されます。

コード・オフセットとデータ・オフセットを指定するには、メイン・メニューから「File」→「Load Symbols」→「Load Symbols with Offsets」の順に選択し、「Load Symbols」ダイアログ・ボックスから目的の COFF ファイルを選択します。COFF ファイルが選択されると、コードとデータが開始する実際のアドレスを入力するための「Load Symbols with Offsets」ダイアログ・ボックスが表示されます。

図 5-6. データ・オフセット



シンボル・ファイル内のすべてのシンボルが指定の値で自動的にオフセットされます。

5.1.6.2 シンボル情報のみの追加

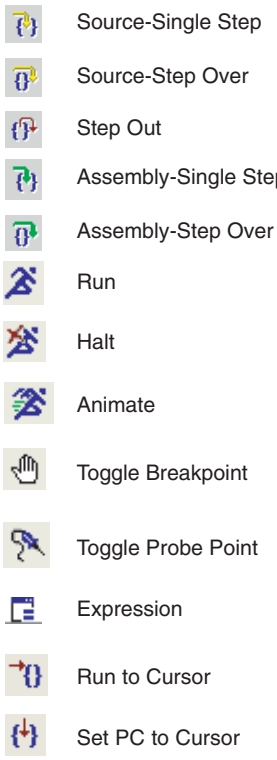
シンボル情報を既存のシンボル・テーブルに追加することもできます。このコマンドは、新規シンボルをロードする前に既存のシンボル・テーブルがクリアされないという点において、「Load Symbol」コマンドとは異なります。

シンボル情報を追加する手順は、シンボルをロードする前述の手順と同様です。オフセット付きの場合は、「File」→「Load Symbols」→「Add Symbols with Offsets」の順に、オフセットなしの場合は、「File」→「Load Symbols」→「Add Symbols Only」の順に、それぞれ選択します。

5.2 基本的なデバッグ

Code Composer Studio IDE には、基本的なデバッグに使用可能なコンポーネントが用意されています。次の図では、CCStudio 内でデバッグに使用されるアイコンを示しています。これらのアイコンがツールバーに表示されていない場合、「View」→「Debug Toolbars」→「ASM/Source Stepping」の順に選択します。「Debug Toolbars」オプション・リストから、表示可能なデバッグ用の各種ツールバーのリストを確認します。表示可能なツールバーには、メニュー内の名前の横にチェックマークが付いています。

図 5-7. 実行時およびデバッグ時に使用されるツールバーのアイコン



5.2.1 実行 / ステップ実行

5.2.1.1 実行

プログラムを実行するには、IDE のメニュー上にある「Debug」項目の下の適切なコマンドを選択します。「Target Control」ツールバーが表示されている場合、「Run」アイコンは左側の縦に並んだツールバー上に表示されます。これらのアイコンがまだ表示されていない場合は、「View」→「Debug Toolbars」→「Target Control」の順に選択します。

次のコマンドを使用して、プログラムを実行します。

- **Main**
  main でデバッグを開始するには、「Debug」→「Go Main」の順に選択します。これにより、main 関数が実行されます。
- **Run**
  実行が停止した後に実行を続行するには、「Run」ボタンを押します。
- **Run to Cursor**
  プログラムを特定の場所まで実行するには、カーソルを目的の場所に移動してから、このボタンをクリックします。
- **Set PC to Cursor**
  プログラム・カウンタを特定の場所に設定するには、カーソルを目的の場所に移動してから、このボタンをクリックします。
- **Animate**
  ブレークポイントに達するまでプログラムを実行させます。ブレークポイントでは、実行が停止し、プローブ・ポイントに接続されていないウィンドウがすべて更新されます。プロー

ブ・ポイントは実行を停止させ、プローブ・ポイントに関連付けられているすべてのグラフとウィンドウを更新してから、実行を継続します。このボタンを押すと、実行をアニメーション表示できます。
アニメーション速度を変更するには、「Option」メニューから「Customize」を選択します。

- **Halt**
  最後に、このボタンを押すと、いつでも実行を停止させることができます。

5.2.1.2 ステップ実行

ソースとアセンブリのステップ実行は、実行が停止した場合のみ使用可能です。ソースのステップ実行は、ソース・エディタ内に表示されるコード行を 1 行ずつ実行します。アセンブリのステップ実行は、「Disassembly」ウィンドウ内に表示される命令行を 1 行ずつ実行します。「View」→「Mixed Source/ASM」の順に選択して mixed Source/ASM モードにすると、ソースとアセンブリの両方のコードを同時に見ることができます。

ステップ実行コマンドを実行するには、ツールバー上の適切なステップ実行アイコンを選択します。別の方法で同様の処理を実行するには、「Debug」→「Assembly/Source Stepping」の順に選択し、適切なコマンドを実行します。

ステップ実行には、次の 3 つのタイプがあります。

- Single Step / Step Into は、1 つの文を実行します。その後実行は停止します。
- Step Over は、関数を実行し、その関数から復帰後に停止します。
- Step Out は、現行のサブルーチンを実行し、呼び出し元の関数に復帰します。実行は、呼び出し元の関数に復帰後に停止します。

5.2.1.3 PDM 用のマルチプロセッサ・ブロードキャスト・コマンド

パラレル・デバッグ・マネージャ（PDM）を使用すると、すべての実行 / ステップ実行コマンドは現在のグループ内のすべてのターゲット・プロセッサにブロードキャストされます。デバイス・ドライバが同期動作をサポートしている場合、次に示す各コマンドは、それぞれのプロセッサ上で同時に開始されるように同期がとられます。

- Locked Step/Step Into を使用すると、実行中ではないすべてのプロセッサに対して Single Step を実行できます。
- Step Over を使用すると、実行中ではないすべてのプロセッサに対して Step Over を実行できます。
- プロセッサがすべてサブルーチン内にある場合、Step Out を使用すると、実行中ではないすべてのプロセッサに対して Step Out を実行できます。
- Run は、実行中ではないすべてのプロセッサに対して Run コマンドを送信します。
- Halt は、すべてのプロセッサを同時に停止させます。
- Animate は、実行中ではないすべてのプロセッサに対してアニメーション表示を開始させます。
- Run Free は、プローブ・ポイントを含むブレークポイントをすべて無効にしてから、現在の PC ロケーションからプログラムを実行させます。

5.2.2 ブレークポイント

ブレークポイントは、デバッグ・セッションの重要なコンポーネントです。ブレークポイントはプログラムの実行を停止させます。プログラムが停止すると、プログラム状態の確認、変数の確認または変更、コール・スタックの確認などを行うことができます。ブレークポイントは、「Editor」ウィンドウ内のソース・コードの行か、「Disassembly」ウィンドウ内の逆アセンブルされた命令の行に設定できます。ブレークポイントを設定した後に、それを有効または無効にできます。

ソース・コード行に設定されるブレークポイントの場合、関連付けられている逆アセンブリ・コード行が表示されることが重要になります。コンパイラの最適化がオンになっている場合、多くの

ソース・コード行でブレークポイントを設定できません。設定可能な行を表示するには、「Editor」ウィンドウ内で混合モードを使用します。

注：
Code Composer Studio は、ブレークポイントをソース・ウィンドウ内の有効な行に移動させ、ブレークポイントを設定する行の横にある選択マージン内にブレークポイント・アイコンを置きます。設定可能な行を判別できない場合は、メッセージ・ウィンドウにエラーが報告されます。

注：
Code Composer Studio は、プローブ・ポイントに達すると必ずターゲットを一時的に停止させます。プローブ・ポイントに接続されるウィンドウまたは表示内容は、実行が停止した時点で更新されます。したがって、プローブ・ポイントを使用する場合、ターゲット・アプリケーションは、リアルタイムの制約条件を満たすことができない場合があります。開発のこの段階では、アルゴリズムのテストを行っています。後から、RTDX と DSP/BIOS を使用してリアルタイムの動作を解析できます。

5.2.2.1 ソフトウェア・ブレークポイント

ブレークポイントは、「Disassembly」ウィンドウまたは C/C++ ソース・コードを含むドキュメント・ウィンドウ内に設定できます。設定できるソフトウェア・ブレークポイント数は、書き込み可能なメモリ・ロケーション（RAM）に設定する限り、制限されていません。ブレークポイント命令を目的の位置に追加するようにターゲット・プログラムを変更することで、ソフトウェア・ブレークポイントは動作します。

ソフトウェア・ブレークポイントを設定する手順は、次のとおりです。

1. ドキュメント・ウィンドウまたは「Disassembly」ウィンドウ内で、ブレークポイントを設定する行にカーソルを移動させます。
2. ドキュメント・ウィンドウの場合、行の先頭にある選択マージン内をダブルクリックします。「Disassembly」ウィンドウの場合、目的の行をダブルクリックします。

選択マージン内にブレークポイント・アイコン（赤一色のドッド）がある場合には、目的の位置にブレークポイントが設定されていることを示しています。

「Toggle Breakpoint」コマンドまたは「Toggle Breakpoint」ボタンを使用しても、ブレークポイントの設定と解除を行うことができます。

1. ドキュメント・ウィンドウまたは「Disassembly」ウィンドウ内で、ブレークポイントを設定する行にカーソルを移動します。
2. 右クリックして「Toggle Breakpoint」を選択するか、「Project」ツールバー上の「Toggle Breakpoint」アイコンをクリックします。

5.2.2.2 ハードウェア・ブレークポイント

ハードウェア・ブレークポイントは、ソフトウェア・ブレークポイントとは異なり、ターゲット・プログラムを変更しません。ハードウェア・ブレークポイントは、チップ上で使用可能なハードウェア・リソースを使用します。ハードウェア・ブレークポイントは、ROM メモリ内にブレークポイントを設定したり、命令取得時ではなくメモリ・アクセス時に中断する場合に有効です。ブレークポイントは、特定のメモリ読み取り、メモリ書き込み、またはメモリ読み取り / 書き込みに対して設定できます。メモリ・アクセスに対するブレークポイントは、ソース・ウィンドウやメモリ・ウィンドウには表示されません。使用可能なハードウェア・ブレークポイント数は、DSP ターゲットによって異なります。

ハードウェア・ブレークポイントにはカウンタを設定することもできます。このカウンタを使用すると、ある位置に指定した回数だけ到達するとブレークポイントが生成されるように設定する

こともできます。カウントが 1 の場合、ブレークポイントは毎回生成されます。ハードウェア・ブレークポイントは、シミュレートするターゲット上では実装できません。

ハードウェア・ブレークポイントを設定する手順は、次のとおりです。

1. 「Debug」→「Breakpoints」の順に選択します。「Break/Probe Points」ダイアログ・ボックスで「Breakpoints」タブが選択された状態で表示されます。
2. 「Breakpoint type」フィールドで、命令取得ブレークポイントの場合は「H/W Break」を選択し、メモリ・アクセス・ブレークポイントの場合は「Break on <bus> <Read|Write|R/W> at location」を選択します。
3. ブレークポイントを設定するプログラムまたはメモリの位置を入力します。次のいずれかの方法を使用します。
   - 絶対アドレスの場合は、任意の有効な C 式、C 関数の名前、またはシンボル名を入力できます。
   - C ソース・ファイルに基づきブレークポイントの位置を入力します。この方法は、C 命令が実行可能プログラムのどこに対応するかがわからない場合に有効です。C ソース・ファイルに基づき位置を指定する場合のフォーマットは、次のとおりです。
     fileName line lineNumber（ファイル名 行 行番号）
4. 「Count」フィールドに、ブレークポイントが生成されるまでに繰り返されるその位置の検出回数を入力します。毎回中断する場合は、1 に設定します。
5. 「Add」ボタンをクリックして、新規にブレークポイントを作成します。これにより、新規ブレークポイントが作成され、有効になります。
6. 「OK」をクリックします。

5.2.3 プローブ・ポイント

5.2.3.1 プローブ・ポイント機能

プローブ・ポイントは、PC 上のファイルからデータを読み取ります。プローブ・ポイントは、アルゴリズム開発に便利なツールです。プローブ・ポイントは、次の用途に使用できます。

- アルゴリズムで使用するためにホスト PC 上のファイルからターゲット上のバッファに入力データを転送する。
- 解析用にターゲット上のバッファからホスト PC 上のファイルに出力データを転送する。
- グラフなどのウィンドウをデータで更新する。

5.2.3.2 プローブ・ポイントとブレークポイントとの違い

プローブ・ポイントとブレークポイントは、両方ともターゲットを停止してから処理を実行するという点では同じです。ただし、プローブ・ポイントは次の点でブレークポイントとは異なります。

- プローブ・ポイントは、ターゲットを一時的に停止させ、1 つの処理を実行し、ターゲットの実行を再開させます。
- ブレークポイントは、実行が手作業で再開されるまで、且つ開いているウィンドウがすべて更新されるまで CPU を停止します。
- プローブ・ポイントでは自動的にファイル入力または出力を実行できますが、ブレークポイントではできません。

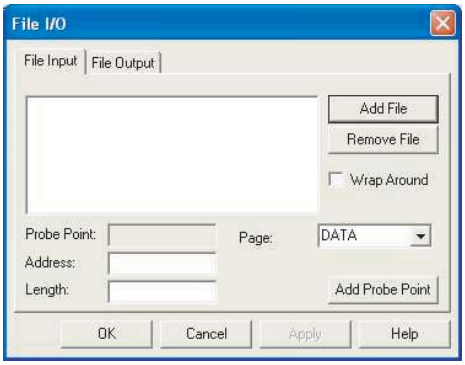
5.2.3.3 プローブ・ポイントを使用した PC ファイルからターゲットへのデータ転送

ここでは、プローブ・ポイントを使用して、PC ファイルの内容をテスト・データとしてターゲットに転送する方法について説明します。プローブ・ポイントに達したときに開いているすべてのウィンドウを更新するために、ブレークポイントも使用します。

1. 「File」→「Load Program」の順に選択します。filename.out を選択し、「Open」をクリックします。
2. 「Project View」内で、filename.c ファイルをダブルクリックします。

3. プローブ・ポイントを追加する main 関数の行にカーソルを移動します
4. 「Toggle Software Probe Point」ツールバー・ボタンをクリックします。
5. 「File」メニューから、「File I/O」を選択します。入力ファイルと出力ファイルを選択する「File I/O」ダイアログ・ボックスが表示されます。

図 5-8. 「File I/O」ダイアログ・ボックス

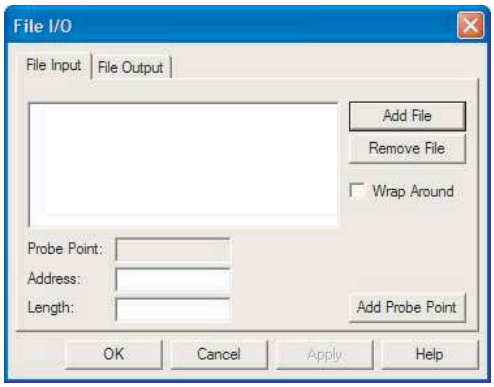


6. 「File Input」タブで、「Add File」をクリックします。
7. プロジェクト・フォルダを指定し、filename.dat を選択して「Open」をクリックします。filename.dat ファイルに対応する制御ウィンドウが表示されます。プログラムを実行するとき、このウィンドウを使用して、データ・ファイル内で開始、停止、巻き戻し、早送りを実行できます。

図 5-9. データ・ファイルの制御

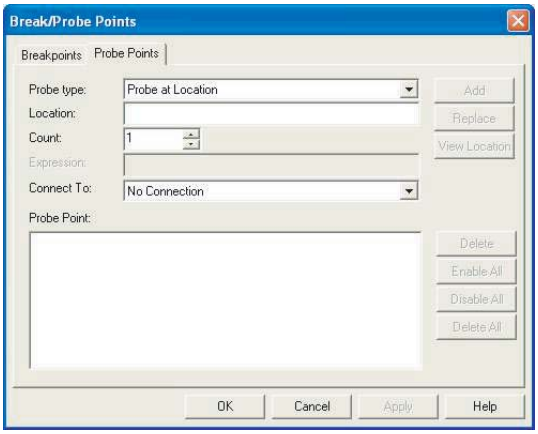
8. 「File I/O」ダイアログ・ボックスで、「Address」と「Length」の値をそれぞれ変更します。また、「Wrap Around」チェックボックスにチェックマークを付けます。「Address」フィールドで、ファイルからデータを入れる場所を指定します。「Length」フィールドで、プローブ・ポイントに到達するたびにデータ・ファイルから読み込むサンプル数を指定します。「Wrap Around」オプションを使用すると、データがファイルの終わりに達したときにファイルの先頭からデータの読み込みを開始できます。これにより、データ・ファイルが連続したデータ・ストリームとして処理されます。

図 5-10. ファイルの追加



9. 「Add Probe Point」をクリックします。「Break/Probe Points」ダイアログ・ボックスの「Probe Points 」タブが表示されます。

図 5-11. 「Probe Point」タブ



10. 「Probe Point」リストから、前に作成した「Probe Point」を選択します。
11. 「Connect To」フィールドで、下向き矢印をクリックしてリストから .dat ファイルを選択します。
12. 「Replace」をクリックします。「Probe Point」リストが変更され、このプローブ・ポイントが filename.dat ファイルに接続されていることが示されます。
13. 「OK」をクリックします。「File I/O」ダイアログ・ボックスに、ファイルがプローブ・ポイントに接続されていることが示されます。
14. 「OK」をクリックして、「File I/O」ダイアログ・ボックスを閉じます。

5.2.4 「Watch」ウィンドウ

5.2.4.1 「Watch」ウィンドウを使用した変数の値の追跡

プログラムをデバッグするときに、プログラム実行中に変数の値がどのように変更されるかを理解することは役に立つ場合があります。「Watch」ウィンドウでは、ローカル変数とグローバル変数の値、および C/C++ の式を監視することができます。「Watch」ウィンドウの詳細は、オンライン・ヘルプの「Watch」ウィンドウに関するトピックを参照してください。「Help」→「Contents」→「Debugging Window & Analysis Tools」→「Debugging Window」→「Watch Window」の順に選択します。

「Watch」ウィンドウを開く手順は、次のとおりです。

1. 「View」→「Watch Window」の順に選択するか、「Watch」ツールバーから「Watch Window」ボタンをクリックします。「Watch」ウィンドウには、「Watch Locals」と「Watch 1」の 2 つのタブがあります。
   - 「Watch Locals」タブ内には、現在実行中の関数に対してローカルな変数の Name（名前）、Value（値）、Type（型）、Radix（基数）が自動的に表示されます。
   - 「Watch 1」タブ内には、ユーザーの指定するローカル変数、グローバル変数および式の Name（名前）、Value（値）、Type（型）、Radix（基数）が表示されます。
2. 「File」→「Load Program」の順に選択します。
3. 「Project View」内で、filename.c ファイルをダブルクリックします。
4. ブレークポイントを設定する行にカーソルを置きます。
5. 「Toggle Breakpoint」ツールバー・ボタンをクリックするか、F9 を押します。選択マージンにブレークポイントが設定されたことが示されます（赤のアイコン）。
6. 「View」→「Watch Window」の順に選択します。ウィンドウの右下隅に別の領域が表示されます。実行時にはこの領域に監視されている変数の値が表示されます。デフォルトでは、「Watch Locals」タブが選択され、実行中の関数に対してローカルな変数が表示されます。
7. main にいない場合には、「Debug」→「Go Main」の順に選択します。
8. 「Debug」→「Run」の順に選択するか、F5 キーを押すか、Run アイコンをクリックします。「Watch」ウィンドウで、ローカルな値が更新されます。

図 5-12. 「Watch Locals」タブ

9. 「Watch 1」タブを選択します。
10. 「Name」列内にある「Expression」アイコンをクリックし、監視対象の変数の名前を入力します。
11. 変更内容を保存するには、「Watch」ウィンドウ内の余白部分をクリックします。次の例のように、ただちに値が表示されます。

図 5-13. 監視対象変数の指定

12. 「Step Over」ツールバー・ボタンをクリックするか、F10 を押して、監視する変数への呼び出しをステップ・オーバーします。

単純変数の値を監視するだけでなく、構造体の要素の値を監視することもできます。

5.2.4.2 「Watch」ウィンドウを使用した構造体の要素の値の監視

構造体の要素の値を監視する手順は、次のとおりです。

1. 「Watch 1」タブを選択します。
2. 「Name」列内にある「Expression」アイコンをクリックし、監視対象の式の名前を入力します。
3. 変更内容を保存するには、「Watch」ウィンドウ内の余白部分をクリックします。
4. + 記号を 1 回クリックします。行が拡張され、構造体の要素とその値がすべて表示されます(「Link」に表示されるアドレスは、変わる場合があります)。

図 5-14. 要素変数の監視

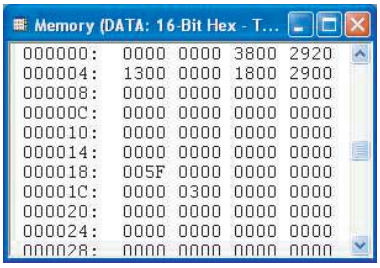
5. 構造体の要素の「Value」をクリックし、その要素の値を編集します。
6. 変数の値を変更します。

「Watch」ウィンドウ内で値が変わっていることに注意してください。この値の色は赤に変わり、値が手作業で変更されたことを示します。

5.2.5 「Memory」ウィンドウ

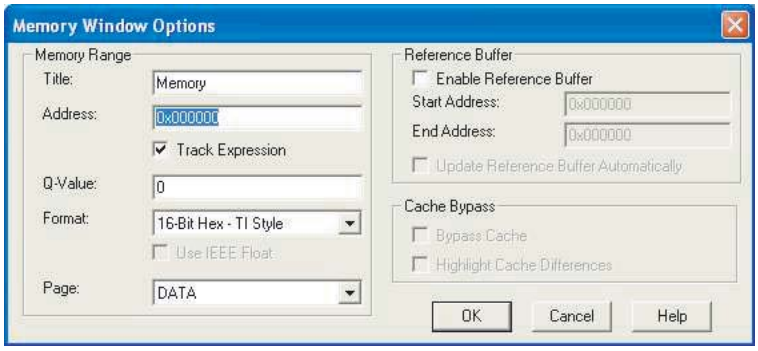
「Memory」ウィンドウを使用すると、指定されたアドレスで始まるメモリの内容を表示できます。オプションを使用して、「Memory」ウィンドウの表示形式を変更できます。また、選択されたメモリ位置の内容を編集することもできます。

図 5-15. 「Memory」ウィンドウ



「View」→「Memory」の順に選択して表示される「Memory Window Options」ダイアログ・ボックスでは、「Memory」ウィンドウのさまざまな特性を指定できます。

図 5-16. 「Memory Window Options」ダイアログ・ボックス



「Memory Window Options」ダイアログ・ボックスには、次のオプションがあります。

- **Title**
  「Memory」ウィンドウの名前を入力します。「Memory」ウィンドウが表示されるときに、この名前がタイトル・バーに表示されます。この機能は、複数の「Memory」ウィンドウを表示させる場合に特に有効です。
- **Address**
  表示するメモリ位置の開始アドレスを入力します。
- **Track Expression**
  このオプションにチェックマークを付けると、「Memory」ウィンドウは開始アドレスに関連付けられている式に基づき、自動的に開始アドレスを再度評価し、変更します。ターゲットが停止、再起動、式のシンボルの変更のいずれかを行うと、「Memory」ウィンドウが再度評価され、再配置されます。たとえば、SP で「Memory」ウィンドウを開いた場合、コードのステップスルー、新しいプログラムのロード、レジスタ自体の変更のいずれかを行うと、このウィンドウは何度も再配置されます。
- **Q-Value**
  Q 値を使用した整数を表示できます。この値は、より正確な 2 進値として整数値を表すために使用されます。小数点がその 2 進値に挿入され、最下位ビット（LSB）からのオフセットは、次のように Q 値によって決定されます。
  $New\_integer\_value = integer / 2^{Q\ value}$
  Q 値が xx ということは、小数点が最下位ビット（LSB）から xx 桁変位した符号付き 2 の補数の整数であることを示します。
- **Format**
  ドロップダウン・リストから、メモリ表示形式を選択します。さまざまな表示形式の詳細は、オンライン・ヘルプを参照してください。
- **Enable Reference Buffer**
  後で比較するときに使用できるように指定されたメモリ領域のスナップショットを保存します。
- **Start Address**
  リファレンス・バッファに保存するメモリ位置の開始アドレスを入力します。このフィールドは、「Enable Reference Buffer」にチェックマークが付いている場合にのみアクティブになります。
- **End Address**
  リファレンス・バッファに保存するメモリ位置の終了アドレスを入力します。このフィールドは、「Enable Reference Buffer」にチェックマークが付いている場合にのみアクティブになります。

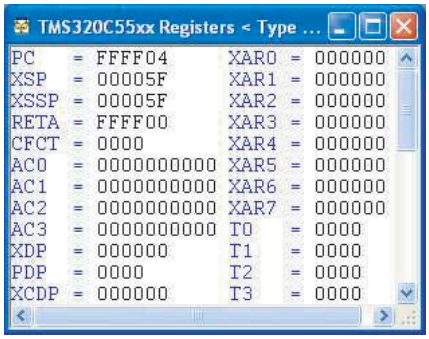
- **Update Reference Buffer Automatically**
  このチェックボックスにチェックマークが付いていると、リファレンス・バッファの内容が、指定されたアドレス範囲の現在のメモリ内容で自動的に上書きされます。このオプションにチェックマークが付いている場合、「Memory」ウィンドウが更新されるたびにリファレンス・バッファが更新されます（たとえば、「Refresh Window」が選択されたり、ブレークポイントに到達したり、ターゲットの実行が停止したりする場合）。このチェックボックスにチェックマークが付いていない場合、リファレンス・バッファの内容は変更されません。このオプションは、「Enable Reference Buffer」にチェックマークが付いている場合にのみアクティブになります。
- **Bypass Cache**
  このオプションを使用すると、常に物理メモリからメモリ内容を読み込みます。通常、メモリの内容がキャッシュ内にあった場合、戻されるメモリ値には物理メモリからではなく、キャッシュからの値が表示されます。このオプションにチェックマークが付いている場合、Code Composer Studio はキャッシュされたメモリ内容を無視するか、バイパスします。
- **Highlight Cache Differences**
  このオプションを使用すると、キャッシュされた値と物理的な値が異なる場合にメモリ位置の値が強調表示されます。また、このキャッシュの相違を強調表示するために、複数の色を使用することもできます。「Option」→「Customize」→「Color」の順に選択し、「Screen Element」ドロップダウン・ボックスの下から「Cache Bypass Differences」オプションを選択します。

詳細は、オンライン・ヘルプから「Memory Window」に関するトピックを参照してください。

5.2.6 「Register」ウィンドウ

「Register」ウィンドウを使用すると、ターゲット上のさまざまなレジスタの内容を表示したり編集したりできます。

図 5-17. 「Register」ウィンドウ



「Register」ウィンドウにアクセスするには、「View」→「Registers」の順に選択し、表示または編集するレジスタ・セットを選択します。

レジスタ内容をアクセスするには、「Edit」→「Register」の順に選択するか、「Registers」ウィンドウからレジスタをダブルクリックするか、「Register」ウィンドウ内で右クリックしてから「Edit Register」を選択します。

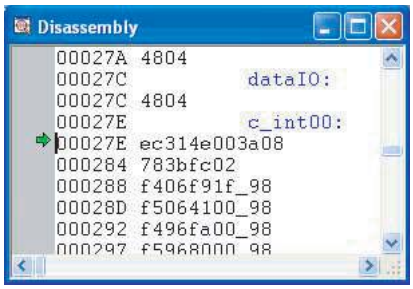
図 5-18. レジスタ変数値の編集

5.2.7 逆アセンブリ / 混合モード

5.2.7.1 逆アセンブリ・モード

プログラムを実際のターゲットまたはシミュレートするターゲットにロードすると、デバッガは自動的に「Disassembly」ウィンドウを開きます。

図 5-19. 「Disassembly」ウィンドウ



「Disassembly」ウィンドウには、デバッグに必要な逆アセンブルされた命令とシンボル情報が表示されます。逆アセンブリは、アセンブリ・プロセスを逆に実行し、メモリの内容をアセンブリ言語コードとして表示します。シンボル情報は、ターゲット上のアドレスまたは値を示すシンボルと英数字の文字列から構成されています。

ステップ実行コマンドを使用してプログラムをステップスルーすると、PC は次の有効な命令まで進みます。

5.2.7.2 混合モード

デバッガは、「Disassembly」ウィンドウ内で逆アセンブルされた命令を表示するだけでなく、逆アセンブルされたコードが差し込まれた C ソース・コードを表示することもできます。モードを変更するには、「View」→「Mixed Source/ASM」の順に選択して切り替えるか、ソース・ウィンドウ内で右クリックして「Mixed Mode」（混合モード）または「Source Mode」（ソース・モード）を選択します。

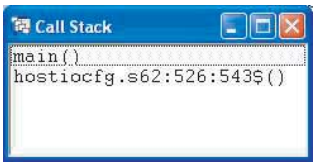
5.2.8 コール・スタック

「Call Stack」ウィンドウを使用すると、デバッグ中のプログラム内の現在の位置に至るまでの一連の関数呼び出しを調べることができます。

「Call Stack」を表示する手順は、次のとおりです。

1. 「View」→「Call Stack」の順に選択するか、「Debug」ツールバーから「View Stack」ボタンをクリックします。

図 5-20. 「Call Stack」ウィンドウ



2. 「Call Stack」ウィンドウに表示されている関数をダブルクリックします。この関数を含むソース・コードが、ドキュメント・ウィンドウに表示されます。目的の関数内の現在実行中の行にカーソルが設定されます。「Call Stack」ウィンドウ内で関数を選択すると、その関数の範囲内にあるローカル変数を確認することができます。

コール・スタックは、C プログラムでのみ動作します。呼び出し元の関数は、ランタイム・スタック上でフレーム・ポインタのリンク・リストを辿って決定されます。プログラムには Stack セクションと main 関数が必要です。これらがない場合、コール・スタックには「C source is not available」（C ソースは使用できません）というメッセージが表示されます。「Call Stack」ウィンドウには、出力の先頭から 100 行のみが表示されていることにも注意してください。

5.2.9 シンボル・ブラウザ

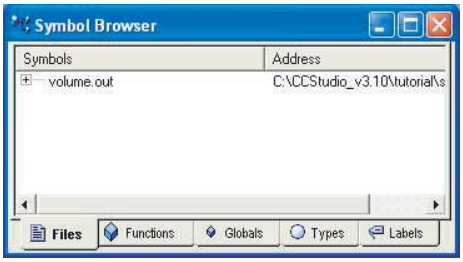
「Symbol Browser」ウィンドウ（図 5-21）には、ロードされた COFF 出力ファイル（*.out）に対して 5 つのタブが付いたウィンドウが表示されます。

- All associated files（すべての関連付けられたファイル）
- Functions（関数）
- Global variations（グローバル変数）
- Types（型）
- Labels（ロードされた COFF 出力ファイル（*.out）に含まれるラベル）

それぞれのタブ付きウィンドウには、さまざまなシンボルを表すノードが含まれています。ノードの先頭にあるプラス記号（+）は、そのノードが展開可能であることを示します。ノードを展開するには、+ 記号をクリックします。展開されたノードの先頭には、マイナス記号（−）があります。そのノードの内容を非表示にするには、−記号をクリックします。

「Symbol Browser」ウィンドウを開くには、「View」→「Symbol Browser」の順に選択します。

図 5-21. 「Symbol Browser」ウィンドウ



シンボル・ブラウザ・ツールの詳細は、オンライン・ヘルプのシンボル・ブラウザに関するトピックを参照してください。「Help」→「Contents」→「Debugging Window & Analysis Tools」→「Debugging Window」→「Symbol Browser」の順に選択します。

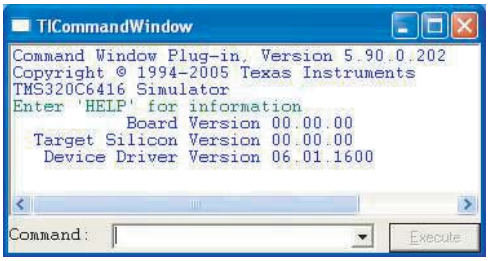
5.2.10 「Command」ウィンドウ

「Command」ウィンドウを使用すると、TI デバッガのコマンド構文を使用してデバッガに対するコマンドを指定できます。

多くのコマンドは C 式のパラメータに対応しています。このため、命令セットは比較的小さく、かつ強力なものにできます。一部の型の C 式の計算が既存の値に影響を与えることがあるので、同じコマンドを使用して値を表示したり、変更したりできます。

「Command」ウィンドウを開くには、「Tools」→「Command Window」の順に選択します。

図 5-22. 「Command」ウィンドウ



「Command」ウィンドウの詳細は、オンライン・ヘルプの「Command」ウィンドウに関するトピックを参照してください。「Help」→「Contents」→「Debugging Window & Analysis Tools」→「Debugging Window」→「Command Window」の順に選択します。

5.3 デバッグ拡張機能

5.3.1 拡張イベント・トリガ（AET）

拡張イベント・トリガ（AET）は、イベント解析およびイベント・シーケンサの両ツールを使用して、ハードウェア解析を単純化します。

イベント解析は、単純なインターフェイスを使用して、ジョブと呼ばれる共通ハードウェア・デバッグ・タスクを構成する作業を支援します。ブレークポイント、アクション・ポイント、カウンタを設定するには、コンテキスト・メニューを使用して、ドラッグ・アンド・ドロップします。イベント解析を使用するには、「Tools」→「Advance Event Triggering」→「Event Analysis」の順に選択します。

イベント・シーケンサは、ターゲット・プログラム内の条件を検索し、これらの条件を満たす内容が検出されたときに特定の処理を開始します。CPU が停止しているときに、ユーザーは条件と処理を指定してから、ターゲット・プログラムを実行させます。シーケンス・プログラムは、指定された条件を検索し、要求された処理を実行します。

5.3.1.1 イベント解析

イベント解析を使用すると、次のジョブを実行できます。

- ブレークポイントの設定
  - ハードウェア・ブレークポイント
  - カウント付きハードウェア・ブレークポイント
  - チェーン・ブレークポイント
  - グローバル・ハードウェア・ブレークポイント
- アクション・ポイントとウォッチ・ポイントの設定
  - データ・アクション・ポイント
  - プログラム・アクション・ポイント
  - ウォッチポイント
  - データ付きウォッチポイント
- カウンタの設定
  - データ・アクセス・カウンタ
  - プロファイル・カウンタ

- ウォッチドッグ・タイマ
- 汎用カウンタ

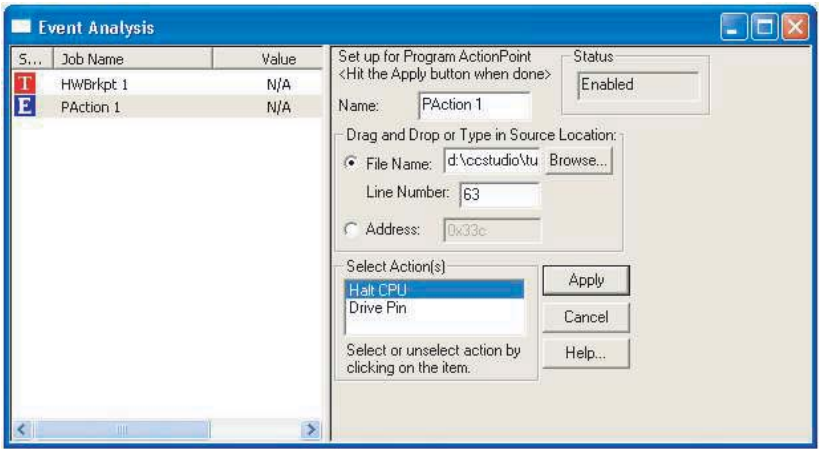
- その他
  - ここまでのベンチマーク
  - エミュレーション・ピンの構成

イベント解析の詳細は、オンライン・ヘルプのイベント解析に関するトピックを参照してください。「Help」→「Contents」→「Debugging Window & Analysis Tools」→「Analysis Tools」→「Advanced Event Triggering」の順に選択します。

イベント解析を使用するジョブを構成するには、Code Composer Studio IDE がオンチップ解析機能を含むターゲット・プロセッサ用に構成されている必要があります。イベント解析を使用するには、「Tools」メニューから「Event Analysis」を選択します。一旦ジョブを構成すると、ターゲット上でコードを実行する際にジョブが有効になり、解析が実行されます。構成済みのジョブを有効または無効にする方法については、オンライン・ヘルプから「Advanced Event Triggering」を参照してください。

1. 「Tools」→「Advanced Event Triggering」→「Event Analysis」の順に選択すると、「Event Analysis」ウィンドウが表示されます。

図 5-23. 「Event Analysis」ウィンドウ



2. 「Event Analysis」ウィンドウ内で右クリックし、「Event Triggering」→「Job Type」→「Job」の順に選択します。ジョブ・メニューは、ターゲット構成に従って動的にビルドされます。ターゲットでサポートされていないジョブは、グレーで表示されます。
3. 「Job」ダイアログ・ボックスに、情報を入力します。
4. 「Apply」をクリックしてジョブをプログラムし、変更内容を保存します。

5.3.1.2 イベント・シーケンサ

イベント・シーケンサは、ターゲット・プログラム内の条件を検索し、これらの条件を満たす内容が検出されたときに特定の処理を開始します。CPU が停止しているときに、ユーザーは条件と処理を指定してから、ターゲット・プログラムを実行させます。シーケンス・プログラムは、指定された条件を検索し、要求された処理を実行します。

イベント・シーケンサの詳細は、オンライン・ヘルプのイベント・シーケンサに関するトピックを参照してください。「Help」→「Contents」→「Debugging Window & Analysis Tools」→「Analysis Tools」→「Advanced Event Triggering」の順に選択します。

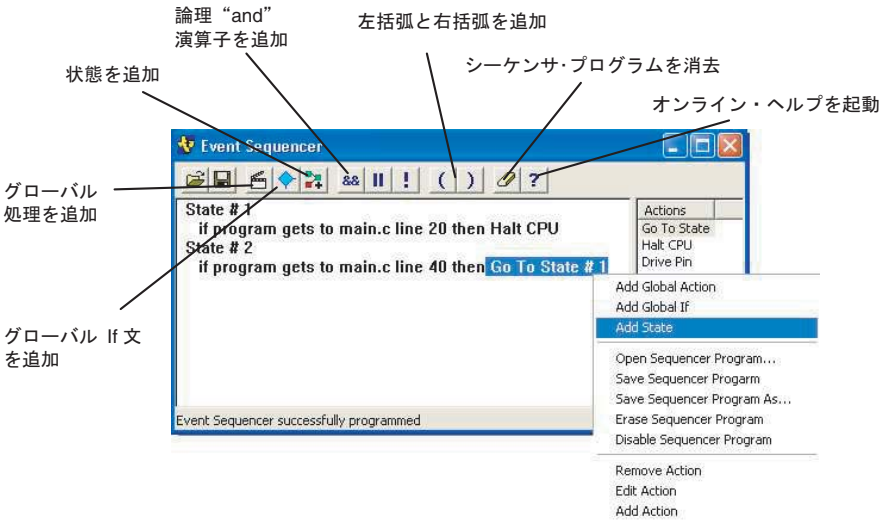
イベント・シーケンサを使用する場合は、Code Composer Studio IDE がオンチップ解析機能を含むターゲット・プロセッサ用に構成されている必要があります。イベント・シーケンサを使用する

には、「Tools」メニューから「Event Sequencer」を選択します。一旦イベント・シーケンサプログラムを作成すると、ターゲット上でコードを実行するときにプログラムが有効になり、解析が実行されます。イベント・シーケンサ・プログラムの作成に関する詳細は、オンライン・ヘルプから「Advanced Event Triggering」を参照してください。

イベント・シーケンサを有効にする手順は、次のとおりです。

1. 「Tools」→「Advanced Event Triggering」→「Event Sequencer」の順に選択します。「Event Sequencer」ウィンドウが表示されます。

図 5-24. 「Event Sequencer」ウィンドウ



2. 「Event Sequencer」ウィンドウ内で右クリックするか、「Event Sequencer」ツールバー・ボタンを使用して、シーケンサ・プログラムを作成します。

5.4 リアルタイム・デバッグ

従来のデバッグ手法（停止モード）では、プログラムはシステムを完全に停止させ、つまりすべてのスレッドを停止させ、割り込みが処理されないようにする必要がありました。システムまたはアプリケーションに、リアルタイム制約がない限り、停止モードをデバッグ専用で使用できます。しかし、アプリケーションの実際のシステム動作を確認したい場合、Code Composer Studio IDE には、リアルタイム・モード（5.4.1 項を参照）、rude リアルタイム・モード（5.4.2 項を参照）、RTDX（5.4.3 項を参照）などのオプションが用意されています。

5.4.1 リアルタイム・モード

リアルタイム・モードを使用すると、ターゲットがバックグラウンド・コード内で停止しているときに、フォアグラウンド・コード内でタイムクリティカルな割り込みを行うことができます。タイムクリティカルな割り込みは、バックグラウンド・コードが停止しているときでも処理する必要がある割り込みです。たとえば、タイムクリティカルな割り込みは、モーター・コントローラまたは高速タイマを処理します。プログラム実行を複数の場所で一時停止させることができます。これにより、他の割り込みを処理しながら、1 つのタイムクリティカルな割り込みを処理することができます。

リアルタイム・モードのデバッグを有効にする手順は、次のとおりです。

1. 割り込みを構成し、ブレークポイントを設定して、リアルタイム・モードの準備をする必要があります。詳細は、オンライン・チュートリアルの「Real-Time Emulation」に関するトピックを参照してください。「Debug」→「Real-time Mode」の順に選択します。制御ウィンドウの一番下にあるステータス・バーには、「POLITE REALTIME」と表示されます。

2. リアルタイムのリフレッシュ・オプションを構成するには、「View」→「Real-Time Refresh Options」の順に選択します。最初のオプションは、「Watch」ウィンドウを更新する頻度を指定します。「Global Continuous Refresh」チェックボックスにチェックマークを付けると、ターゲット・データを表示するすべてのウィンドウ（Memory/Graph/Watch ウィンドウなど）が連続して更新されます。ある特定のウィンドウだけを連続して更新するには、このチェックボックスのチェックマークを外し、特定のウィンドウのコンテキスト・メニューから「Continuous Refresh」を選択します。
3. 「OK」をクリックして、ダイアログ・ボックスを閉じます。
4. 「View」→「Registers」→「Core Registers」の順に選択して、「Core Register」ウィンドウを開きます。デバッグ・インターラプト・イネーブル・レジスタ（DIER）がリスト内に表示されます。DIER を使用して、インターラプト・イネーブル・レジスタ（IER）によって、ユーザーが選択する単一の、または特定のサブセットの、またはすべての割り込みをリアルタイム（タイムクリティカル）割り込みとして指定します。DIER は、体系的に指定した IER をミラーリングします。
5. レジスタを右クリックして、「Edit Register」を選択します。
6. リアルタイム割り込みとして設定する割り込みを指定する新しいレジスタ値を入力します。
7. 「Done」をクリックして、「Edit Registers」ダイアログ・ボックスを閉じます。
8. リアルタイム・モード・デバッグが構成されます。

5.4.2 rude リアルタイム・モード

優先順位の高い割り込み、または他のコード・セクションが、極めてタイムクリティカルになることがあります。これらを実行するために必要なサイクル数は、最小値または決まった数に留める必要があります。これは、デバッグ処理（実行制御とレジスタ / メモリ・アクセスの両方）が一部のコード領域内で禁止されているか、特定のマシン状態をターゲットとする必要があることを意味します。デフォルトのリアルタイム・モードでは、プロセッサは権限のない「polite」モードで稼働しています。つまり、デバッグ処理は、適切な処理の遅延を受け入れ、デバッグ依存のウィンドウには干渉しません。

しかし、デバッグ・コマンド（実行制御とレジスタ / メモリ・アクセスの両方）は、デバッグ処理依存ではないウィンドウを特定できない場合に失敗することがあります。デバッガを制御可能にするために、リアルタイム・デバッグを「polite」から「rude」モードに変更する必要があります。rude リアルタイム・モードでは、権限があるため、デバッグ処理はデバッグ・アクセスを禁止する保護を変更して、遅延することなく正常に実行されます。また、クリティカルなコード領域は、デバッグ処理が rude リアルタイム・モードに切り替わるまでデバッグできません。

rude リアルタイム・モードを有効にするには、次の手順のいずれかを実行します。

- デバッグ・コマンドが失敗したときに、表示ウィンドウから「Perform a Rude Retry」を選択します。
- リアルタイムが有効になっているときに、「Debug」メニューから「Enable Rude Real-time Mode」を選択します。

rude リアルタイム・モードが有効になっていると、メイン・プログラム・ウィンドウの一番下にあるステータス・バーに「RUDE REALTIME」と表示されます。rude リアルタイム・モードを無効にするには、「Debug」メニューの「Enable Rude Real-time Mode」の選択を解除します。ステータス・バーには、「POLITE REALTIME」と表示されます。

rude リアルタイム・モードが有効になっているときに CPU を停止する場合、デバッグ・アクセスがブロックされている、つまりタイムクリティカルな ISR 用のときでさえ CPU が停止する可能性があります。ユーザーがブレークポイントに応答するまで、CPU は何も実行することができないので、CPU は適切な時間内にその ISR を完了できません。この問題を回避するには、「Enable Rude Real-time Mode」の選択を解除して、polite リアルタイム・モードに戻る必要があります。

リアルタイム・モードの詳細は、オンライン・ヘルプのリアルタイム・デバッグに関するトピックを参照してください。「Help」→「Contents」→「Debugging Window & Analysis Tools」→「Analysis Tools (Generic)」→「Real Time Debugging」の順に選択します。

5.4.3 リアルタイム・データ・エクスチェンジ（RTDX）

DSP/BIOS リアルタイム解析（RTA）機能は、リアルタイム・データ・エクスチェンジ（RTDX）リンクを利用して、ターゲット・データをリアルタイムで取得し監視します。RTDX リンクを利用すると、RTDX API ライブラリを使用して、DSP ターゲットに対する独自のカスタマイズしたインターフェイスを作成できます。

リアルタイム・データ・エクスチェンジ（RTDX）は、ターゲット・アプリケーションに干渉することなく、ホスト・コンピュータとターゲット・デバイス間でデータを転送します。この双方向通信パスにより、ホストがデータ収集だけでなく、ホストが実行中のターゲット・アプリケーションに相互作用することも可能になります。ターゲットから収集したデータは、ホスト上で解析し、視覚化することができます。アプリケーション・パラメータは、アプリケーションを停止させることなく、ホスト・ツールを使用して調整できます。また、RTDX を使用すると、ホスト・システムはターゲット・アプリケーションとアルゴリズムに対してデータをシミュレートすることもできます。

RTDX は、ターゲットとホストの両方のコンポーネントから構成されます。小さい RTDX ソフトウェア・ライブラリがターゲット・アプリケーション上で実行されます。ターゲット・アプリケーションは、このライブラリの API に対して関数呼び出しを行い、データの受け渡しを行います。このライブラリはスキャンベースのエミュレータを使用して、JTAG インターフェイスによりホスト・プラットフォームとのデータをやり取りします。ターゲット・アプリケーション実行時、ホストへのデータ転送はリアルタイムで行われます。

ホスト・プラットフォーム上で、RTDX ホスト・ライブラリは Code Composer Studio IDE と連携して動作します。データの視覚化と解析ツールは、COM API を介して RTDX と通信して、ターゲット・データの取得および DSP アプリケーションへのデータ送信の両方またはいずれか一方のみ実行します。

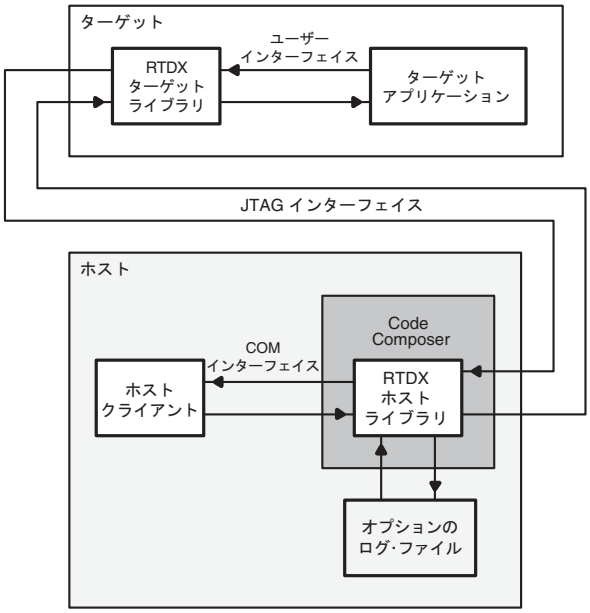
ターゲット・アプリケーションからデータを受信するモードとして、ホスト・ライブラリがサポートしているのは、連続モードと非連続モードの 2 つのモードです。連続モードでは、データは RTDX ホスト・ライブラリによりバッファリングされるだけで、ログ・ファイルには書き込まれません。連続モードは、開発者がターゲット・アプリケーションからデータを連続的に取得して表示し、さらにデータをログ・ファイルに保存する必要がない場合に使用します。非連続モードでは、データはホスト上のログ・ファイルに書き込まれます。このモードは、量に限りのあるデータを取得し、そのデータをログ・ファイルに記録する場合に使用します。

RTDX の使用方法については、オンライン・ヘルプの「Help」→「Contents」→「Real-Time Data Exchange (RTDX)」、または「Help」→「Tutorial」→「RTDX」を参照してください。

5.4.3.1 RTDX データ・フロー

RTDX は、ターゲット・アプリケーションとホスト・クライアント間の両方向のデータ・パイプから形成されています。このデータ・パイプは、次に示すようにハードウェア・コンポーネントとソフトウェア・コンポーネントの組み合わせから構成されています。

図 5-25. RTDX データ・フロー



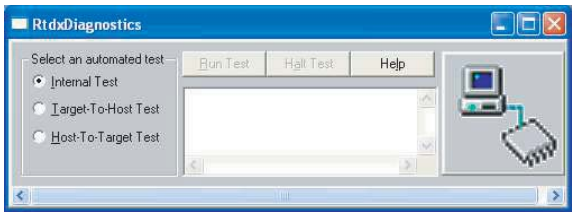
5.4.3.2 RTDX をグラフィカルに構成

RTDX ツールを使用すると、RTDX をグラフィカルに構成でき、RTDX チャネルの設定、RTDX の診断の実行を行うことができます。これらのツールにより、データ転送時の RTDX 機能を拡張できます。

RTDX には、3 つのメニュー・オプションとして Diagnostics Control（診断制御）、Configuration Control（構成制御）、Channel Viewer Control（チャネル・ビューア制御）があります。

Diagnostics Control（診断制御）
RTDX は、システム上で RTDX が正常に動作していることを確認するための RTDX 診断制御を備えています。この診断テストは、ターゲットからホストへの転送、およびホストからターゲットへの転送における基本機能をテストします。RTDX 診断制御を開くには、「Tools」→「RTDX」→「Diagnostics Control」の順に選択します。このテストは、RTDX が有効になっている場合にのみ使用できます。

図 5-26. 「RTDX Diagnostics」ウィンドウ

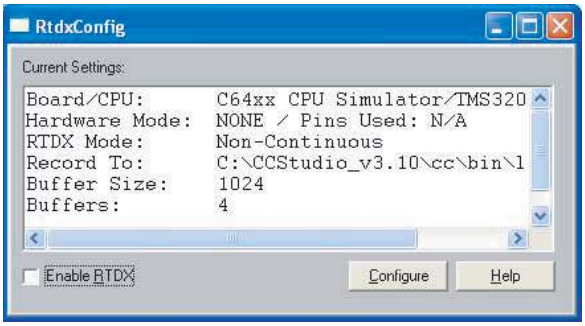


Configuration Control（構成制御）
構成制御は、RTDX のメイン・ウィンドウです。このウィンドウで実行できる内容は、次のとおりです。

- 現在の RTDX 構成内容を表示する
- RTDX を有効または無効にする
- 「RTDX Configuration Control Properties」ページにアクセスして、RTDX を再構成し、ポート構成の設定値を選択する

RTDX 構成制御を開くには、「Tools」→「RTDX」→「Configuration Control」の順に選択します。

図 5-27. 「RTDX Configuration」ウィンドウ

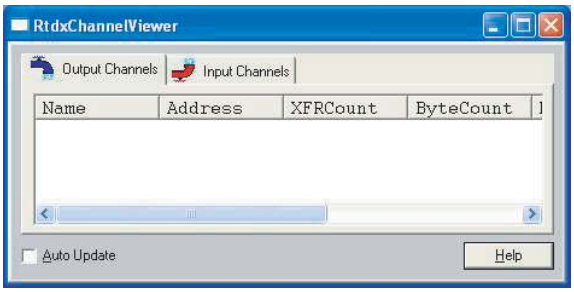


Channel Viewer Control（チャネル・ビューア制御）
RTDX チャネル・ビューア制御は、ターゲットで宣言されたチャネルを自動的に検出し、検出したチャネルを表示可能リストに追加する Active X コントロールです。RTDX チャネル・ビューア制御が実行できる内容は、次のとおりです。

- ターゲットで宣言されたチャネルを表示可能リストから削除または追加する
- リスト上のチャネルを有効または無効にする

RTDX チャネル・ビューア制御を開くには、「Tools」→「RTDX」→「Channel Viewer Control」の順に選択します。「Channel Viewer Control」ウィンドウが表示されます。

図 5-28. 「RTDX Channel Viewer」ウィンドウ



「Input Channel」タブと「Output Channel」タブをクリックすると、それぞれのチャネルのリストを表示します。「Input Channel」ウィンドウと「Output Channel」ウィンドウでは両方とも、チャネルを表示、削除、再度追加できます。

「Auto-Update」にチェックマークを付けると、表示を更新しなくても、すべてのチャネルの情報が自動的に更新されます。「Auto-Update」機能を使用していない場合は、ウィンドウ内で右クリックして、コンテキスト・メニューから「Refresh」を選択してすべてのチャネルの情報を更新します。

> **注：**
> RTDX チャネル・ビューア制御で特定のチャネルの拡張チャネル情報を受信するためには、RTDX クライアントは関心のあるチャネルを開いておく必要があります。

5.4.3.3 1 つの整数値をホストに転送

RTDX の基本機能は、1 つの整数値をホストに転送することです。次のステップでは、ターゲットからホストへ、ホストからターゲットへそれぞれデータを転送するプロセスの概要を説明します。異なる型のデータ転送に関する特定のコマンドと詳細については、オンライン・ヘルプを参照し

てください（「Help」→「Contents」→「Real-Time Data Exchange (RTDX)」、または「Help」→「Contents」→「RTDX」の順に選択）。

ターゲット・アプリケーションからホストにデータを転送する手順は、次のとおりです。

1. リアルタイム・データをキャプチャするターゲット・アプリケーションを準備します。特定の RTDX 構文をアプリケーション・コードに挿入して、ターゲットからホストにリアルタイム・データの転送を可能にします。ターゲット・アプリケーションを準備するプロセスはすべてのデータ型で同じですが、データ転送にはデータ毎に異なる関数呼び出しが必要になります。このため、整数の配列をホストに転送するのではなく、単一の整数をホストに転送するだけに特化した関数呼び出しを追加する必要があります。
2. データを処理するホスト・クライアントを準備します。目的のチャネル毎に 1 つの RTDX オブジェクトをインスタンス化し、指定されているオブジェクトのチャネルを開き、その他の目的の関数を呼び出します。
3. Code Composer Studio IDE を起動します。
4. ターゲット・アプリケーションを TI プロセッサにロードします。
5. 「Tools」→「RTDX」→「Configuration Control」の順に選択して表示される「Enable RTDX」チェックボックスにチェックマークを付けます。
6. ターゲット・アプリケーションを実行してリアルタイム・データをキャプチャし、そのデータを RTDX ホスト・ライブラリに転送します。
7. ホスト・クライアントを実行してデータを処理します。

RTDX の使用方法については、オンライン・ヘルプを参照してください（「Help」→「Contents」→「Real-Time Data Exchange (RTDX)」、または「Help」→「Tutorial」→「RTDX」の順に選択）。

5.4.3.4 ホストからのデータ受信

クライアント・アプリケーションは、データをターゲットに書き込むことにより、ターゲット・アプリケーションにデータを転送できます。クライアント・アプリケーションからターゲットに転送されたデータは、最初に RTDX ホスト・ライブラリでバッファリングされます。このデータは、ターゲットからのデータの要求が到着するまで RTDX ホスト・ライブラリに保持されます。RTDX ホスト・ライブラリにその要求を満たす十分なデータがある場合、ターゲット・アプリケーションに干渉することなく、そのデータをターゲットに書き込みます。

バッファの状態は、変数バッファの状態に戻されます。正の値は、ターゲットがまだ要求していない、RTDX ホスト・ライブラリでバッファリングされたバイト数を示します。負の値は、RTDX ホスト・ライブラリがまだ満たしていない、ターゲットから要求されたバイト数を示します。

ホスト・クライアントからターゲット・アプリケーションにデータを転送する手順は、次のとおりです。

1. データを受信するターゲット・アプリケーションを準備します。ホスト・クライアントからデータを読み込む、単純な RTDX ターゲット・アプリケーションを作成します。
2. データを転送するホスト・クライアントを準備します。目的のチャネル毎に 1 つの RTDX オブジェクトをインスタンス化し、指定されているオブジェクトのチャネルを開き、その他の目的の関数を呼び出します。
3. Code Composer Studio IDE を起動します。
4. ターゲット・アプリケーションを TI プロセッサにロードします。
5. 「Tools」→「RTDX「→「Configuration Control」の順に選択して表示される「Enable RTDX」チェックボックスにチェックマークを付けます。
6. ターゲット・アプリケーションを実行します。
7. ホスト・クライアントを実行します。

RTDX の使用方法については、オンライン・ヘルプを参照してください（「Help」→「Contents」→「Real-Time Data Exchange (RTDX)」、または「Help」→「Tutorial」→「RTDX」の順に選択）。

5.5 自動化（デバッグ用）

5.5.1 GEL（General Extension Language）の使用

前述のように、GEL スクリプトを使用して、カスタム GEL メニューを作成し、Code Composer Studio の動作を自動化することができます。4.6.1 項では、組み込み GEL 関数を使用して、プロジェクト管理に関連した作業を自動化する方法を説明しています。また、デバッグ・プロセス中の作業を自動化するための組み込み GEL 関数も多数あります。たとえば、GEL スクリプトから実行可能な処理には、ブレークポイントの設定、「Watch」ウィンドウへの変数の追加、実行の開始、実行の停止、ファイル I/O のセットアップなどがあります。

5.5.2 デバッグ用のスクリプト・ユーティリティ

スクリプト・ユーティリティ（4.6.2 項）も、多数のデバッグ作業を自動化できるコマンドを備えています。詳細は、スクリプト・ユーティリティに付属のオンラインヘルプを参照してください。

5.6 リセット・オプション

ターゲットまたはエミュレータのリセットが必要な場合には、Code Composer Studio IDE に用意されているコマンドを使用します。これらのリセット・コマンドが実行できるかどうかは、IDE がターゲットに接続されているかによって異なります。ターゲットとの接続または切断に関する詳細は、3.1.3 項を参照してください。

5.6.1 ターゲットのリセット

ターゲットをリセットすると、すべてのレジスタの内容が起動状態に初期化され、プログラムの実行が停止します。ターゲット・ボードがこのコマンドに応答せず、しかもカーネル・ベースのデバイス・ドライバを使用している場合は、CPU カーネルが機能しないことがあります。この場合は、そのカーネルをリロードする必要があります。

シミュレータは、ターゲット・シミュレーションの指定に従って、すべてのレジスタの内容を起動状態に初期化します。

ターゲット・プロセッサをリセットするには、「Debug」→「Reset CPU」の順に選択します。

> **注：**
> **「Debug」→「Reset CPU」のオプションを使用できるようにするには、ターゲットとの接続を確立する必要があります。**

5.6.2 エミュレータのリセット

一部のプロセッサでは、ハードウェア・リセットを機能させる前に、プロセッサを実行状態にしておく必要があります。この場合、プロセッサを実行状態に置くための唯一の方法は、エミュレータをリセットすることです。エミュレータをリセットすると、TRST ピンがアクティブになり、デバイスが実行モードになります。

「Reset Emulator」オプションは、Code Composer Studio がターゲットから切断されると常に使用できます。エミュレータをリセットするには、「Debug」→「Reset Emulator」の順に選択します。「Reset Emulator」を実行すると、ハードウェアは実行状態に置かれ、リセット・ボタンを押すか、または「Debug」→「Reset CPU」の順に選択して、手作業でターゲット・ハードウェアをリセットできます。ただし、これは ARM デバイスには適用されません。

第 6 章

解析 / チューニング

高機能アプリケーションを作成するには、目的に合わせて性能、パワー、コード・サイズ、コストに重点を置いて開発する必要があります。

アプリケーション・コード解析とは、アプリケーション効率に影響を与える要因に関わるデータを収集して解釈するプロセスです。アプリケーション・コード・チューニングとは、アプリケーション効率を向上させるためにコードに変更を加える作業です。DSP 開発者は、お客様、アプリケーション、ハードウェアが要求する効率性に関する目標を達成するために、何度もアプリケーションの解析を行い、チューニングすることができます。

Code Composer Studio には、アプリケーションの解析とチューニングを支援するさまざまなツールが用意されています。

項目 **ページ**

6.1 アプリケーション・コード解析

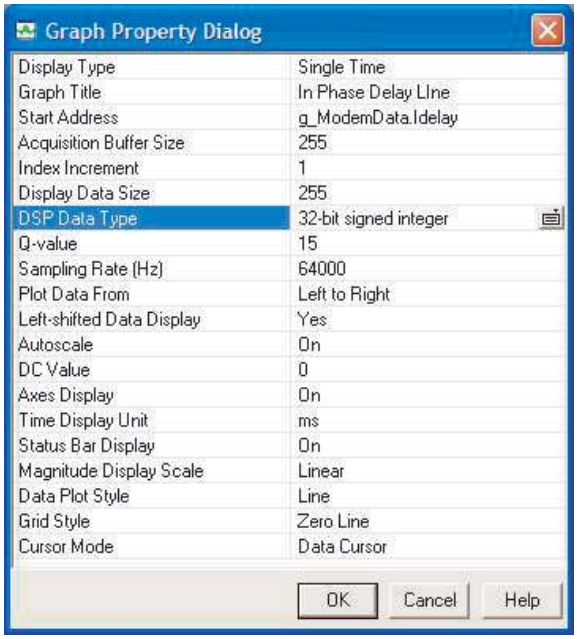
Code Composer Studio の解析ツールは、重要なデータを収集し、そのデータを DSP 開発者が使用できるように設計されています。

6.1.1 データの視覚化

プログラムで処理されるデータをグラフ化する方法が豊富に用意されています。使用可能なグラフは、Time/Frequency、Constellation Diagram、Eye Diagram、Image です。

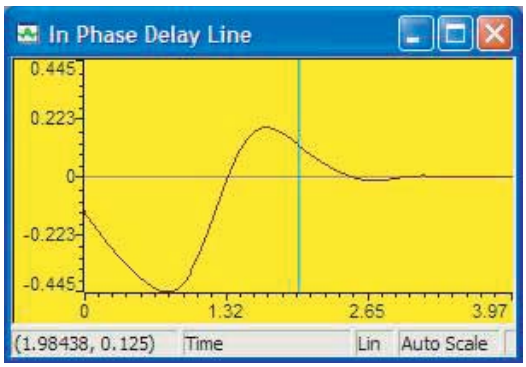
これらのグラフにアクセスするには、「View」→ 「Graph」の順に選択し、目的のグラフを選択します。次に「Graph Property」ダイアログ・ボックスからグラフ・プロパティを指定します。次の例では、Single Time (Time/Frequency) の「Graph Property」ダイアログ・ボックスを示しています。

図 6-1. 「Graph Properties」ダイアログ・ボックスの例



プロパティを構成して、「OK」ボタンをクリックすると、グラフ・ウィンドウが開き、グラフ・プロパティで指定したデータがグラフ表示されます。

図 6-2. グラフの例



詳細は、オンライン・ヘルプの「Graphing Data」に関するトピックを参照してください。

6.1.2 シミュレータ解析

シミュレータ解析ツールは、特定のシステム・イベントについて報告します。このため、プログラムの性能を正確にモニタし、測定することができます。

シミュレータ解析のユーザー・オプションは、次のとおりです。

- 解析を有効または無効にする
- 選択したイベントの発生回数をカウントする
- 選択したイベントが発生するたびに実行を停止する
- カウントを削除またはイベントをブレークする
- ログ・ファイルを作成する
- イベント・カウンタをリセットする

シミュレータ解析ツールを使用する手順は、次のとおりです。

1. プログラムをロードします。
2. 解析ツールを起動します。使用するデバイスに合わせて「Tools」→「Simulator Analysis」の順に選択します。
3. 「Simulator Analysis」ウィンドウ内で右クリックし、コンテキスト・メニューから「Enable Analysis」を選択します（有効になっていない場合）。
4. プログラムを実行またはステップスルーします。
5. 解析ツールの出力を解析します。

シミュレータ解析ツールの詳細は、オンライン・ヘルプの「Simulator Analysis」に関するトピックを参照してください。

6.1.3 エミュレータ解析

エミュレータ解析ツールを使用すると、イベントおよびハードウェア・ブレークポイントの設定、モニタ、およびカウントを行うことができます。

エミュレータ解析ツールを起動するには、プログラムをロードしてから、メニューバーから使用するデバイスに合わせて「Tools」→「Emulator Analysis」の順に選択します。

「Emulator Analysis」ウィンドウには、列ごとに次の情報が表示されます。

- **Event：**イベント名
- **Type：**イベントのタイプ（ブレーク・イベントまたはカウント・イベント）
- **Count：**プログラムが停止するまでに発生したイベントの回数
- **Break Address：**ブレーク・イベントが発生したアドレス
- **Routine：**ブレーク・イベントが発生したルーチン

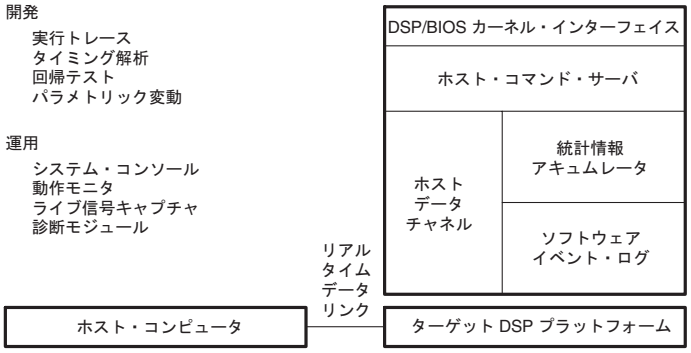
> **注：**
> **プロファイル・クロックを使用しているときは、解析機能は使用できません。**

エミュレータ解析ツールの詳細は、オンライン・ヘルプの「Emulator Analysis」に関するトピックを参照してください。

6.1.4 DSP/BIOS リアルタイム解析（RTA）ツール

DSP/BIOS リアルタイム解析（RTA）機能（図 6-3）を使用すると、実行中の DSP アプリケーションのプローブ、トレース、監視を行うことで、アプリケーションを視覚的に確認できます。これらのユーティリティは、デバッガが使用しているものと同じ物理 JTAG 接続に乗った状態で存在し、この接続をターゲットとホスト間の（リアルタイムではあるが）低速の通信リンクとして使用します。

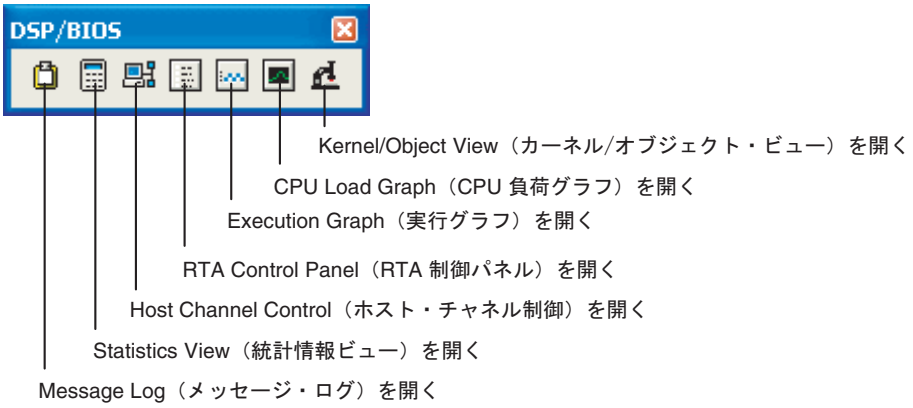
図 6-3. リアルタイム・キャプチャおよび解析



DSP/BIOS RTA では、ターゲット・システム内に DSP/BIOS カーネルが必要です。DSP/BIOS カーネルは、アプリケーションにランタイム・サービスを提供するだけでなく、物理リンクを介してホストとのリアルタイム通信をサポートします。DSP/BIOS API および静的に作成されたオブジェクトを使用してアプリケーションを構築するだけで、リアルタイム情報をキャプチャしてアップロードすることが可能なターゲットを自動的に装備することができます。この情報は、CCStudio の視覚的な解析ツールを駆動します。同様に、補助 API およびオブジェクトを使用すると、ターゲット・プログラムの制御下で明示的な情報をキャプチャすることもできます。保持するユーティリティから見ると、DSP/BIOS はリアルタイムのプログラム解析に対して、広範囲の機能をいくつか提供します。

DSP/BIOS リアルタイム解析ツールは、DSP/BIOS ツールバーからアクセス可能です（図 6-4）。

図 6-4. DSP/BIOS RTA ツールバー



ここでは、このツールバーの各要素について説明します。

- **Message Log**
  独立したリアルタイム・スレッドによってカーネル・ログ・オブジェクトに書き込まれたイベントのシーケンスを時間順に表示します。これは、プログラム全体の制御フローをトレースするときに役立ちます。ターゲット・プログラムがイベントをログに記録する方法は、2 種類あります。
  - 明示的：DSP/BIOS API 呼び出しを使用します。たとえば、「LOG_printf(&trace, “hello world!”)」のように使用します。ここで、「trace」は、ログ・オブジェクトの名前です。
  - 暗黙的：スレッドがレディ、ディスパッチ、終了のときに基本カーネルで実行されます。たとえば、「Execution Graph Details」内にログ・イベントがあります。

  ログをファイルに出力するには、「Message Log」ウィンドウ内で右クリックし、「Property Page」を選択します。

- **Statistics View**
  カーネル・アキュームレータ・オブジェクトに蓄積された統計要約情報を表示します。単純なカウンタと時間に応じて変化するデータ値から、独立スレッドの処理間隔までの動的なプログラム要素が反映されます。ターゲット・プログラムは、スレッドの実行のスケジュール時または I/O オペレーション実行時に、DSP/BIOS API 呼び出しを使用して明示的に、またはカーネルを使用して暗黙的に統計情報を累算します。各種設定内容を変更するには、「Statistics View」ウィンドウ内で右クリックし、「Property Page」を選択します。
- **Host Channel Control**
  プログラムで指定するホスト・チャネルを表示します。このウィンドウを使用して、ファイルをこれらのチャネルにバインドし、チャネル上のデータ転送を開始し、転送されるデータ量を監視できます。カーネル I/O オブジェクトをホスト・ファイルにバインドすると、ターゲット・プログラムにアルゴリズムの決定テスト用の標準データ・ストリームを提供します。カーネル I/O オブジェクトで管理されるその他のリアルタイムなターゲットのデータ・ストリームは、後の解析用にすぐにホスト・ファイルに接続およびキャプチャされます。
- **RTA Control Panel**
  ターゲット・プログラム内のリアルタイム・トレースおよび統計情報の累積を制御します。その結果、これを使用すると、開発者はリアルタイム・プログラムの実行における可視性の度合いを制御することができます。デフォルトでは、すべてのタイプのトレースは有効になっています。すべてのトレース・タイプを有効にするには、「Global host enable」チェックボックスにチェックマークを付ける必要があります。また、プログラムの設定は、このウィンドウ内でも変更できます。「RTA Control Panel」は、「Property Page」に表示される「RTA Control Panel Properties」ダイアログ・ボックスで設定する間隔でプログラムの変化をチェックします。また、「Property Page」では、「Execution Graph」など RTA ツールの更新間隔も変更できます。
- **Execution Graph**
  リアルタイムにスレッドの実行を表示します。「Execution Graph」を使用すると、実行されるスレッドのタイミングとその順序を確認することができます。太い青線は、現在実行中のスレッドを示し、このスレッドが CPU を使用していることを示します。グラフ内の線の情報は、「Execution Graph」ウィンドウ内で右クリックし、「Legend」を選択します。「Message Log」ウィンドウ内で「Execution Graph Details」を表示する場合、「Execution Graph」内のボックス（線のセグメント）をダブルクリックします。ここでは、そのイベントに関する詳細をテキスト形式で表示できます。また、グラフ内のスレッドを非表示にするには、「Execution Graph」ウィンドウ内で右クリックし、「Property Page」を選択します。
- **CPU Load Graph**
  ターゲット CPU が負荷を処理している状況をグラフ表示します。最新の CPU 負荷は、左下隅に表示され、最大の CPU 負荷に達した場合は、右下隅に表示されます。CPU 負荷は、他のスレッドの実行が必要ない場合に実行される、優先順位の低いタスクの処理に費やされない時間と定義されます。このため、CPU の負荷には、ターゲットからホストへデータ転送と追加のバックグラウンド・タスクの実行に必要な時間が含まれています。CPU 負荷は、ポーリング時間の平均値です。ポーリング時間を長くすると、CPU 負荷での瞬間的なスパイクがグラフには表示されないことが多くなります。ポーリング時間を設定するには、「RTA Control Panel」ウィンドウを開いて、そのウィンドウ内で右クリックします。「Property Page」を選択します。次に、「Host Refresh Rates」タブから「Statistics View / CPU Load Graph」スライダーを使用してポーリング時間を設定し、「OK」をクリックします。
- **Kernel/Object View**
  現在ターゲット上で実行中の DSP/BIOS オブジェクトの構成、状態、ステータスを表示します。このツールを使用すると、ターゲット上の動的および静的に構成されたオブジェクトが両方とも表示されます。現在のデータを保存するには、このウィンドウ内で右クリックし、「Save Server Data」を選択します。

注：
ソフトウェア開発時に、DSP/BIOS リアルタイム解析ツールを Code Composer Studio IDE の標準デバッガともに使用すると、プログラム実行中にデバッガが提供するほんのわずかな間隔にターゲット・プログラムの動作を視覚化できます。デバッガがプログラムを停止させてターゲットの制御を得た後でも、DSP/BIOS からキャプチャされた情報は、現在の実行のポイントに達するイベントのシーケンスに対して非常に貴重な予測を提供できます。

また、DSP/BIOS リアルタイム解析ツールは、ハードウェア・ロジック・アナライザと同様に機能するソフトウェアとして役割が増大しています。組み込みの DSP/BIOS カーネルと付属のホスト解析ツールが結びついて、新世代の稼働テストとフィールド診断ツールに対する必要な基盤を形成します。これらのツールは、既存の JTAG 基盤を通じて、稼働中の実システムのアプリケーション・プログラムと対話することができます。

DSP/BIOS の使用に費やすオーバーヘッド・コストは、最小限に抑えられるので、計測をそのまま実行してフィールド診断が可能になります。その結果、開発者は障害を引き起こす実際のデータをキャプチャして解析することができます。

6.1.5 コード・カバレッジとマルチイベント・プロファイラ・ツール

コード･カバレッジとマルチイベント･プロファイラ･ツールは､2 つの異なる機能を備えています。

- コード・カバレッジは、ソース行のカバレッジを視覚化するので、開発者は適切なコード・カバレッジが確保されるようにテストを構築できるようになります。
- マルチイベント・プロファイルを行うと、アプリケーションの一度のシミュレーション実行で指定したイベント、またはすべてのイベントから関数プロファイル・データを収集することができます。イベントには、CPU サイクル、実行される命令、パイプライン・ストール、キャッシュ・ヒット、キャッシュ・ミスなどがあります。このツールを使用すると、性能に影響する可能性のある要因を特定できます。

詳細は、『Code Coverage and Multi-event Profiler User's Guide』(SPRU624) を参照してください。

6.2 アプリケーション・コード・チューニング（ACT）

チューニング・プロセスは、解析終了後に開始します。アプリケーション・コードの解析が完了すると、非効率的なコードが特定されます。チューニング・プロセスは、非効率的なコードを改善できるかどうかを判断し、効率の目標を設定し、およびコードを変更することでこれらの目標の達成を試みるという構成になっています。Code Composer Studio は、複数のツールを連結して 1 つのチューニング・メソッドを構成するだけでなく、チューニングの進行を解析する単一ポイントを提供します。

6.2.1 最適化ダッシュボード

ダッシュボードは、チューニング・プロセスの中心となる部分です。ダッシュボードは、どのチューニング・ツールを使用すべきかについてビルド時および実行時のプロファイル・データに関する提案内容を表示し、各ツールを起動します。開発サイクルのチューニング・フェーズでは、ダッシュボードがメイン・インターフェイスになります。

6.2.1.1.「Advice Window」

「Advice Window」は、チューニング情報を表示するダッシュボードのコンポーネントです。このウィンドウでは、チューニング・プロセスに従ってガイドを表示します。実施すべき適切な手順について詳細に説明し、他のツールや資料へのリンク、および重要なメッセージについて有用な情報を表示します。

ツールを初めて使用する場合、またはチューニング中に実行する処理を適切に判断するために、この「Advice Window」を使用してください。

Code Composer Studio は、最初はデバッグ画面を起動します。「Advice Window」を開くには、ツールバー上にあるフォークの形をした「Tuning Layout」アイコンをクリックしてチューニング画面に切り替えます。別の方法として、「Profile」→「Tuning」メニューから「Advice」を選択することもできます。「Advice Window」は Code Composer Studio 画面の左側に開き、次の Welcome 情報が表示されます。

図 6-5. ダッシュボードのチューニング情報を表示する「Active Window」

「Advice Window」の上部にはツールバーがあり、インターネット操作と同様のメインのアドバイス・ページを開くためのボタンがあります。ツールバー上にある矢印をクリックすると、このタブですでに表示したアドバイス・ページをヒストリ内容に基づき前後に移動して表示することができます。各アドバイス・ページの下部には、1 つまたは複数のタブがあります。これらのタブを用いると、複数のアドバイス・ページを開いたままにできるので、一度に複数の作業を実行することができます。開いているページを切り替えるには、「Advice Window」の下部にあるタブをクリックします。アクティブになっているタブを閉じるには、「Advice Window」内で右クリックし、コンテキスト・メニューから「Close Active Tab」を選択します。

「Welcome」アドバイス・ページには、主なチューニング・ツールの説明へのリンクがあります。「Welcome」アドバイス・ページの一番下にある枠内には、チューニング・プロセスの次のステップに関する情報が表示されます。この青い Action 枠は、「Advice Window」のページ全体を通して表示されます。

「Advice Window」のページを表示している際に、赤い警告メッセージが表示されることがあります。これらのメッセージは、通常、チューニング時の注意事項として表示されますが、途中で遭遇する問題を解決するための重要な情報も提供します。

CodeSizeTune などのツールを起動すると、そのツールのアドバイス・タブが「Advice Window」の下部に表示されます。このページには該当ツールに関する情報が表示されます。ツールの機能やツールを使用するタイミングに関する情報、また最適なチューニングを行うためにそのツールを効果的に適用する方法が表示されます。

6.2.1.2. プロファイル設定

「Advice Window」が示すように、チューニング・プロセスを開始する前にチューニングする目的のコード要素を Code Composer Studio に設定しておく必要があります。この設定を行うには、Profile Setup を使用します。チューニング・プロセスの開始時に「Profile Setup」ウィンドウを使用して、収集するデータと必要なコード・セクションを指定します。

「Profile Setup」ウィンドウは、「Advice Window」を使用して開くことができます。「Welcome」ページの下部にある Action 枠内で、「Setup Advice」ページを開くためのリンクをクリックします。最初の Action 枠内には、「Profile Setup」ウィンドウを開くためのリンクがあります。また、「Profile Setup」ウィンドウは、メイン・メニューから「Profile」→「Setup」の順に選択して、起動することもできます。

「Activities」タブには、プロファイル・セッションがアプリケーション用にデータを収集する動作が表示されます。「Ranges」タブでは、収集すべきデータのプログラム範囲を指定します。チューニング情報を収集する関数、ループ、および任意のコード・セグメントを「Ranges」タブに追加します。「Control」タブを使用して出口ポイントをコードに追加して、Code Composer Studio にデータ収集を停止するタイミングを通知します。「Control」タブでは、収集停止ポイント（Halt）または収集再開ポイント（Resume）をコードに追加して、他のコード・セクションを分離することができます。「Custom」タブを使用して、キャッシュ・ヒットや CPU のアイドル・サイクルなどのカスタム・データを収集します。

6.2.1.3.「Goals」ウィンドウ

アプリケーションをチューニングするには、効率性に関する目標を設定し、それを達成することが必要です。このため、Code Composer Studio は、数値上の目標を記録し、その目標を達成するための進行状況をトラッキングする方法を用意しています。

「Goals」ウィンドウでは、コード・サイズ、サイクル数などを含むアプリケーション・データを表示します。これらの値は、コードが実行されるたびに更新されます。また、このデータは前回実行されたときのデータと比較されるだけでなく、定義した目標とも比較されます。

「Goals」ウィンドウを開くには、「Profile」→「Tuning」メニューから「Goals」を選択します。

図 6-6. 「Goals」ウィンドウ

アプリケーションがロードされていてプロファイルがセットアップされている場合は、アプリケーションを実行するだけで「Goals」ウィンドウにデータが表示されます。チューニング目標を記録するには、「Goals」列内をクリックし、目標を入力して「Enter」を押します。目標が達成されると、そのデータには括弧が付いて緑で表示されます。目標が達成されていない場合、データは赤で表示されます。アプリケーションが再開されて実行されると、「Goals」ウィンドウ内の「Current」の値が「Previous」列に移動し、その差分が「Delta」列に表示されます。また、「Goals」ウィンドウでは、このウィンドウの左側にあるロギング・アイコンを使用して、いつでもログに内容を保存したり、表示したりすることができます。

6.2.1.4.「Profile Viewer」ウィンドウ

「Profile Viewer」ウィンドウは、チューニング・プロセス中に収集されたデータを表示します。「Profile Viewer」ウィンドウは、情報グリッドで構成されます。各行は、「Profile Setup」ウィンドウの「Ranges」タブで選択されるコードの要素に対応しています。グリッド内の列では、「Profile Setup」ウィンドウの「Activities」タブと「Custom」タブで選択されたとおりに、収集されたデータをコードのプロファイル・セクション毎に保存します。

「Profile Viewer」ウィンドウは、チューニング・プロセス中、収集されたすべての情報を表示する唯一の場所です。この情報は、各種データ値でソートできます。「Profile Viewer」ウィンドウに表示されるデータ・セットは、保存、復元、他のデータ・セットとの比較を行うことができます。

「Profile Viewer」ウィンドウを使用すると、最もチューニングを必要とするコード・セクションを正確に特定できます。たとえば、最も多くのキャッシュ・ストールを引き起こす関数を判別するために、「Profile Viewer」ウィンドウ内のキャッシュ・ストールのデータを降順でソートすること

ができます。その関数のセクションのプロファイルを作成し、キャッシュ・ストールの原因となっているコードを正確に特定することができます。

「Profile Viewer」ウィンドウを開くには、メイン・メニューから「Profile」→「Viewer」の順に選択します。別の方法として、「Advice Window」から「Setup」タブに移動します。「Setup Advice」ページの下部にある「Profile Data Viewer」リンクをクリックすると、「Profile Viewer」ウィンドウが画面の下部に表示されます。「Profile Setup」ウィンドウを使用してチューニングをセットアップした場合、アプリケーションを実行するとデータが「Profile Viewer」ウィンドウに表示されます。この表示は、行と列をドラッグ・アンド・ドロップしてカスタマイズすることができます。「Profile Viewer」ウィンドウのボタンを使用して、データの保存および復元を行うことができます。また、列タイトルをダブルクリックすると、データをソートすることができます。さらに複数の「Profile Viewer」ウィンドウを同時に開くこともできます。

6.2.2 コンパイラ・コンサルタント

コンパイラ・コンサルタント・ツールは、C/C++ ソース・コードを解析して、性能を向上させるために、変更すべきコードに関して提案内容を示します。このツールは、2 つのタイプの情報を表示します。コンパイル時ループ情報と実行時ループ情報です。コンパイル時ループ情報は、コンパイラにより作成されます。実行時ループ情報は、アプリケーションをプロファイルすることで収集されるデータです。コードをコンパイルまたはビルドするたびに、コンパイラ・コンサルタントはそのコードを解析し、コード効率を高めるための別の最適化方法を提案します。ユーザーはこのアドバイスを実行し、プロジェクトを再度ビルドすることができます。その後、「Profile Viewer」ウィンドウに、最適化の結果が表示されます。

コンパイラ・コンサルタントの情報を解析する際、次のように列ごとに「Profile Viewer」ウィンドウ内の情報をソートします。

- プロファイルが行われていない場合には、Estimated Cycles Per Iteration をキーにしてソートすると、1 回の繰り返しでどのループのサイクルが最も多くなるかが表示されます（コンパイル時ループ情報）。
- Collect Run Time Loop Information を指定してプロファイルが行われている場合には、cycle.CPU: Excl. Total をキーにソートすると、システムへの影響が考慮されずに、どのループのサイクルが最も多くなるかが表示されます。
- Profile all Functions and Loops for Total Cycles を指定してプロファイルが行われている場合には、cycle.Total: Excl. Total をキーにソートすると、システムへの影響が考慮されて、どのループのサイクルが最も多くなるかが表示されます。
- Advice Count をキーにソートすると、どのループのアドバイスが最も多くなるかが表示されます。

このソートを行うことで、CPU サイクル時間を最も多く消費する行と、チューニングにより最も性能が向上する行が「Profile Viewer」ウィンドウの一番上に表示されます。これにより、1 回に 1 ループのチューニングを行うことができます。任意のループ行の「Advice Types」エントリをダブルクリックすると、そのループに対するアドバイスが「Advice Window」の「Consultant」タブに表示されます。

そのアドバイスに従って個別にループを修正したら、その行を「Profile Viewer」ウィンドウで非表示にすると便利です。行を非表示にすると、「Profile Viewer」ウィンドウに表示される情報量が少なくなります。行は、いつでも再表示することができます。

コンパイラ・コンサルタントの詳細は、オンライン・ヘルプから「Application Code Tuning」下の「Compiler Consultant」を参照してください。

6.2.3 コードサイズ・チューン（CST）

コードサイズ・チューン（CST）ツールを使用すると、アプリケーションのコード・サイズとサイクル・カウント間のトレードオフを簡単に最適化することができます。コードサイズ・チューンは、さまざまなプロファイル構成を使用して、アプリケーションのプロファイルを実施し、個々の関数のデータを収集し、最適なコンパイラ・オプションの組み合わせを決定します。また、CST

はこれらの関数特有のオプション・セットのグラフを生成するので、最適な構成をグラフィカルに選択することが可能になります。

Code Composer Studio の以前からのユーザーは、CST はプロファイル・ベース・コンパイラ（PBC）の代替と考えてください。

1. はじめに、アプリケーションが CST プロファイルの条件に適合しているかどうかを確認するには、オンライン・ヘルプから「Application Code Tuning」下の「CodeSizeTune」を参照してください。
2. CST は、各種プロファイル収集オプション・セットを使用して、アプリケーションのビルドとプロファイルを行います。各種プロファイル収集オプションを使用して取得した関数の性能を示すプロファイル情報を使用して、CST は関数レベルのコンパイラ・オプションを含む収集オプションをつなぎ合わせます。ビルド方法とプロファイル方法の詳細は、オンライン・ヘルプから「CodeSizeTune」下の「Build and Profile」を参照してください。
3. 最適なプロファイル収集オプション・セットが、コード・サイズ対性能の 2 次元グラフに示されます。このグラフでは、システム要件に適合したサイズと速度の最適な組み合わせをグラフィカルに選択できます。適切な収集オプション・セットの選択については、オンライン・ヘルプから「CodeSizeTune」下の「Desired Speed and Code Size」を参照してください。
4. 最後に、選択した収集オプション・セットを Code Composer Studio プロジェクトに保存します。オンライン・ヘルプから「CodeSizeTune」下の「Save Settings and Close」を参照してください。

「Advice Window」では、このプロセスの各ステップを説明しています。コードサイズ・チューンを起動すると、「Advice Window」の「CodeSizeTune」タブが自動的に表示されます。詳細は、オンライン・ヘルプから「CodeSizeTune」の下の「Advice Window」を参照してください。

図 6-7. 「CodeSizeTune」タブに表示されるアドバイス

6.2.4 キャッシュ・チューン

キャッシュ・チューン・ツールを使用すると、一定時間のキャッシュ・アクセスをグラフィカルに視覚化することができます。このツールは、非効率的なキャッシュ使用部分（競合するコード配置や非効率のデータ・アクセス・パターンなどによる）を特定する場合に非常に効果的です。このツールを使用すると、キャッシュ効率を大幅に最適化することができます。その結果、メモリ・サブシステムで消費されるサイクル数が減少し、アプリケーション全体のキャッシュ効率を向上させることができます。

すべのメモリ・アクセスは、タイプ毎に色分けされます。フィルタ、パンニング、ズームの各種機能は、特定の領域をすばやく詳細に表示することができます。このキャッシュ・アクセスの

visual/temporal 表示を使用すると、競合、容量、強制失敗などの問題領域をすばやく特定することができるようになります。

メニューから「Tuning」→「CacheTune」の順に選択すると、キャッシュ・チューン・ツールが起動し、最新のキャッシュ・トレースが表示されます。「Profile Setup」からキャッシュ・トレースを表示するには、各種オプションを選択する必要があります。詳細は、オンライン・ヘルプを参照してください。次の 3 種類のキャッシュ・トレース・ファイルがあります。

- プログラム・キャッシュ・トレース
- データ・キャッシュ・トレース
- クロス・キャッシュ・トレース

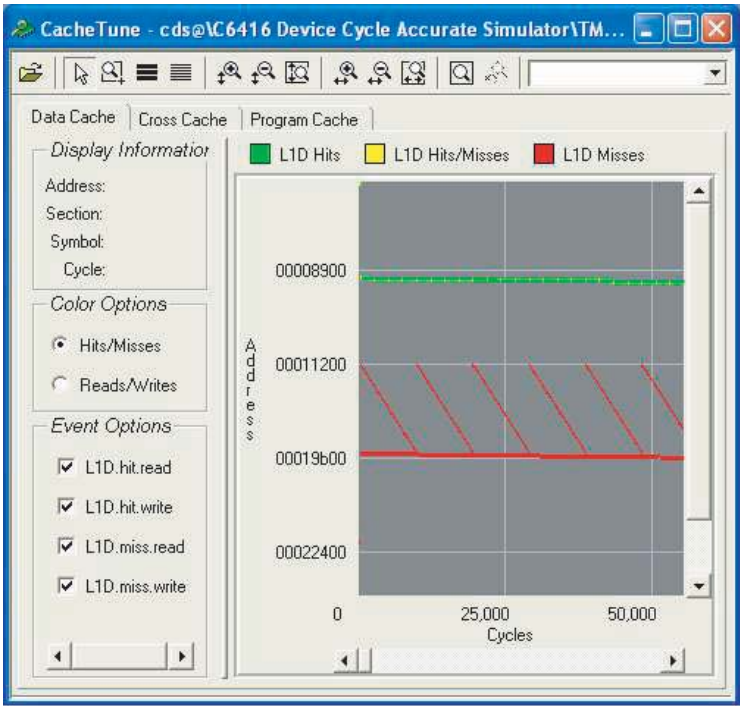
「Data Cache Trace」タブが、デフォルトで表示されます。キャッシュ・トレースを収集していない場合、グラフには何も表示されません。

ツールが起動したら、保存されているデータ・セットを開いて他のキャッシュ・データ・ファイルを表示できます。

データセットを開くには、「Open Dataset」ボタンをクリックするか、そのホットキーを押すか、コンテキスト・メニューから「Load dataset item」をクリックします。

Cache Analysis ツールの詳細は、TI の Web サイトから『Cache Analysis User's Guide』(SPRU575) を参照してください。

図 6-8. キャッシュ・チューン・ツール



第 7 章

その他のツール、ヘルプ、ヒント

この章では、IDE インストレーションのカスタマイズ方法、インストレーション内容の更新方法、その他のヘルプおよび資料の検索方法について説明します。

項目 **ページ**

7.1 コンポーネント・マネージャ

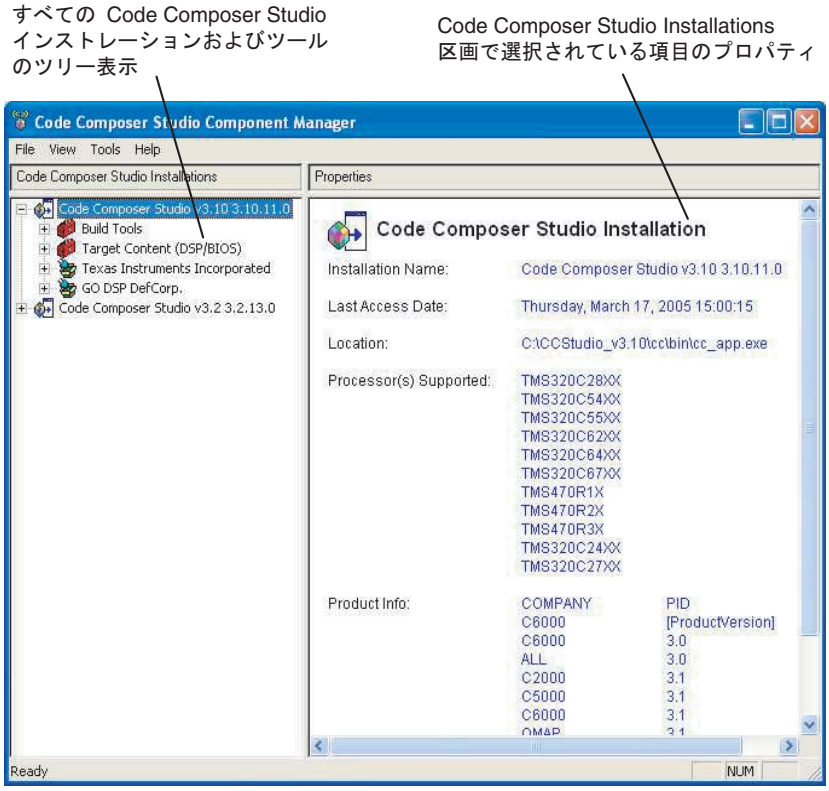
> **注：**
> コンポーネント・マネージャは、主にインストレーション内容のカスタマイズまたは変更に使用される高機能ツールです。このツールは、カスタムまたは複数のインストレーション環境におけるコンポーネントの相互作用を解決する場合にのみ使用してください。

複数の Code Composer Studio IDE をインストールした環境では、インストールされているツールを共用することができます。コンポーネント・マネージャは、このような複数の Code Composer Studio IDE をインストールした環境でバージョンの異なる複数のツールを扱えるようにするインターフェイスを提供します。

「Component Manager」ウィンドウには、すべてのインストレーション、ビルド・ツール、TI 製のプラグイン・ツール、サードパーティ製のプラグイン・ツールのリストが表示されます。ツリー（コンポーネント・マネージャの左の区画）内でノードを選択すると、そのプロパティが「Properties」区画（右の区画）に表示されます（図 7-1 を参照）。

コンポーネント・マネージャを使用すると、特定の Code Composer Studio インストレーション用のツールを有効または無効にすることができます。この機能により、IDE 内に含まれているツールの組み合わせをカスタマイズすることができます。また、コンポーネント・マネージャでは、アップデート・アドバイザにアクセスして、Web から最新バージョンのツールをダウンロードできます。

図 7-1. コンポーネント・マネージャ



7.1.1 コンポーネント・マネージャを開く

コンポーネント・マネージャを開く手順は、次のとおりです。

1. Code Composer Studio IDE 内の「Help」メニューから「About」を選択します。「About Code Composer Studio」ダイアログ・ボックスが表示されます。
2. 「About」ダイアログ・ボックスで、「Component Manager」ボタンをクリックします。「Code Composer Studio Component Manager」ウィンドウが表示されます。

7.1.2 複数のバージョンの Code Composer Studio IDE

複数のバージョンの Code Composer Studio IDE および関連ツールを保守するための要件は、次のとおりです。

- 複数のバージョンの Code Composer Studio IDE または関連ツールを保持するには、各バージョンを別々のディレクトリにインストールする必要があります。
- Code Composer Studio IDE またはツールの別のバージョンを以前のインストレーションと同じディレクトリにインストールすると、オリジナルのインストレーションは上書きされます。
- 1 つのインストレーションで、同じツールの複数のバージョンを有効にすることはできません。

7.2 アップデート・アドバイザ

アップデート・アドバイザを使用すると、更新されたバージョンの Code Composer Studio IDE および関連ツールをダウンロードすることができます。アップデート・アドバイザは、「Available Updates」Web サイトにアクセスします。このサイトには、ダウンロード可能なパッチ、ドライバ、およびツールが表示されます。

アップデート・アドバイザを使用するには、インターネットに接続できるようになっていて、コンピュータにはブラウザがインストールされている必要があります。完全なシステム要件については、オンライン・ヘルプのトピック「Update Advisor」を参照してください。

> **注：**
> 「Available Updates」Web サイトにアクセスするには、my.TI に登録する必要があります。

7.2.1 アップデート・アドバイザの登録

インストール時に製品を登録しなかった場合には、Code Composer Studio のヘルプ・メニューからオンライン登録フォームにアクセスすることができます （「Help」→「Register」の順に選択）。

> **注：**
> アップデート・アドバイザを初めて使用するとき、my.TI の Web ページがブラウザに表示されることがあります。登録するには、このページに表示される指示に従ってください。

アップデート・アドバイザを使用してダウンロードする内容を受け取るには、オンライン登録を行い、有効なアップデート・サービス・プランを用意する必要があります。Code Composer Studio 製品では、90 日間の無償アップデート・サービスを受けることができます (通常は年間アップデート・サービスがバンドルされています)。この期間が終了したときには、年間アップデート・サービスを購入する必要があります。年間アップデート・サービスは、完全な製品でのみ利用可能です。

7.2.2 ツールの更新のチェック

Code Composer Studio IDE から、「Help」→「Update Advisor」→「Check for Updates」の順に選択します。すでに my.TI に登録済みで、自動ログインに必要な cookie を受け入れていれば、「Available Updates」Web サイトがブラウザに直接表示されます。「Available Updates」Web サイトを照会するために、アップデート・アドバイザはご使用のコンピュータから次の特定の情報を渡します。

- Code Composer Studio IDE 製品の登録番号
- Code Composer Studio IDE インストレーション・バージョン
- インストール済み製品の説明
- インストール済みプラグインのリスト

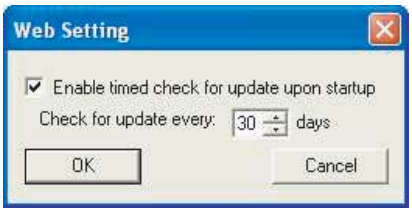
「Available Updates」Web サイトには、ご使用の Code Composer Studio インストレーションに適用可能な更新内容が表示されます。更新内容をダウンロードするだけにするか、またはダウンロードしてから直ちにインストールするかを選択できます。また、自動的に更新をチェックするようにアップデート・アドバイザを構成することもできます。

7.2.3 ツールの更新の自動チェック

ツールの更新は、いつでもチェックすることができます。また、自動的に更新をチェックするようにアップデート・アドバイザを構成することもできます。

1. 「Help」→「Update Advisor」→「Settings」の順に選択します。「Web Setting」ダイアログ・ボックスが表示されます。

図 7-2. アップデート・アドバイザの Web 設定



2. 自動更新機能を有効にするには、「Enable timed check for update upon startup」チェックボックスにチェックマークを付けます。このフィールドを有効にすると、アップデート・アドバイザは指定されたスケジュールに従って自動的に Web の更新をチェックします。
3. 「Check for Update」フィールドで、アップデート・アドバイザが「Available Updates」Web サイトをチェックする頻度を指定します。
4. 「OK」をクリックして変更内容を保存し、ダイアログ・ボックスを閉じます。

7.2.4 更新のアンインストール

インストールされた更新をアンインストールして、以前のバージョンの Code Composer Studio IDE を復元することができます。

直前のバージョンのツールにだけ復元できることに注意してください。ツールに 1 つの更新をインストールしてから、同じツールに 2 つ目の更新をインストールした場合、最初の更新後の状態へは復元できます。2 つ目の更新と最初の更新を両方ともアンインストールしても、オリジナルのバージョンのツールには復元できません。

7.3 その他のヘルプ

「Help」→「Contents」の順に選択して、各トピックの説明を順を追って参照することができます。また、オンライン・チュートリアルを実行したり、オンライン・ヘルプに記述されている Web サイトから最新のヘルプ・トピックを表示したり、特定の機能またはプロセスの情報を記載した PDF 形式のユーザー・マニュアルを参照することもできます。また、アップデート・アドバイザを使用して最新の機能を取得するには、「Help」→「Update Advisor」の順に選択します。

7.3.1 オンライン・ヘルプ

オンライン・ヘルプは、チュートリアル、マルチメディア・デモ、ユーザー・マニュアル、アプリケーション・レポート、ソフトウェアに関する情報を取得することができる Web サイト（www.dspvillage.ti.com）へのリンクを提供しています。「Help」をクリックして、表示されている

リンクを参照します。文脈依存ヘルプを利用するには、IDE の関連部分を選択して、F1 をクリックします。

7.3.2 オンライン・チュートリアル

Code Composer Studio IDE のチュートリアルには、すぐに Code Composer Studio IDE を使い始めることができるようにするレッスンが組み込まれています。Code Composer Studio IDE のチュートリアルを使用するには、「Help」→「Tutorial」の順に選択します。

日本テキサス・インスツルメンツ株式会社

本　　社　〒160-8366　東京都新宿区西新宿６丁目２４番１号　西新宿三井ビルディング３階　　０３（４３３１）２０００（番号案内）
☎
西日本ビジネスセンター　〒530-6026　大阪市北区天満橋１丁目８番３０号　ＯＡＰオフィスタワー26階　　０６（６３５６）４５００（代　　表）
工　　場　大分県・日出町／茨城県・美浦村
研究開発センター　茨城県・つくば市（筑波テクノロジー・センター）／神奈川県・厚木市（厚木テクノロジー・センター）

■お問い合わせ先
プロダクト・インフォメーション・センター（PIC）~~URL: http: //www.tij.co~~.jp/pic/

Code Composer Studio
入門マニュアル

第 5 版 2005 年 6 月
第 4 版 2004 年 10 月
第 3 版 2004 年 10 月
第 2 版 2003 年 3 月
第 1 版 2001 年 9 月

発行所 **日本テキサス・インスツルメンツ株式会社**
〒160-8366
東京都新宿区西新宿 6-24-1（西新宿三井ビルディング）

Made in Japan
’05.06

日本テキサス・インスツルメンツ株式会社

JAJU071

ご注意

日本テキサス・インスツルメンツ株式会社（以下TIJといいます）及びTexas Instruments Incorporated（TIJの親会社、以下TIJないしTexas Instruments Incorporatedを総称してTIといいます）は、その製品及びサービスを任意に修正し、改善、改良、その他の変更をし、もしくは製品の製造中止またはサービスの提供を中止する権利を留保します。従いまして、お客様は、発注される前に、関連する最新の情報を取得して頂き、その情報が現在有効かつ完全なものであるかどうかご確認下さい。全ての製品は、お客様とTIJとの間に取引契約が締結されている場合は、当該契約条件に基づき、また当該取引契約が締結されていない場合は、ご注文の受諾の際に提示されるTIJの標準販売契約約款に従って販売されます。

TIは、そのハードウェア製品が、TIの標準保証条件に従い販売時の仕様に対応した性能を有していること、またはお客様とTIJとの間で合意された保証条件に従い合意された仕様に対応した性能を有していることを保証します。検査およびその他の品質管理技法は、TIが当該保証を支援するのに必要とみなす範囲で行なわれております。各デバイスの全てのパラメーターに関する固有の検査は、政府がそれ等の実行を義務づけている場合を除き、必ずしも行なわれておりません。

TIは、製品のアプリケーションに関する支援もしくはお客様の製品の設計について責任を負うことはありません。TI製部品を使用しているお客様の製品及びそのアプリケーションについての責任はお客様にあります。TI製部品を使用したお客様の製品及びアプリケーションについて想定されうる危険を最小のものとするため、適切な設計上および操作上の安全対策は、必ずお客様にてお取り下さい。

TIは、TIの製品もしくはサービスが使用されている組み合せ、機械装置、もしくは方法に関連しているTIの特許権、著作権、回路配置利用権、その他のTIの知的財産権に基づいて何らかのライセンスを許諾するということは明示的にも黙示的にも保証も表明もしておりません。TIが第三者の製品もしくはサービスについて情報を提供することは、TIが当該製品もしくはサービスを使用することについてライセンスを与えるとか、保証もしくは是認するということを意味しません。そのような情報を使用するには第三者の特許その他の知的財産権に基づき当該第三者からライセンスを得なければならない場合もあり、またTIの特許その他の知的財産権に基づきTI からライセンスを得て頂かなければならない場合もあります。

TIのデータ・ブックもしくはデータ・シートの中にある情報を複製することは、その情報に一切の変更を加えること無く、かつその情報と結び付られた全ての保証、条件、制限及び通知と共に複製がなされる限りにおいて許されるものとします。当該情報に変更を加えて複製することは不公正で誤認を生じさせる行為です。TIは、そのような変更された情報や複製については何の義務も責任も負いません。

TIの製品もしくはサービスについてTIにより示された数値、特性、条件その他のパラメーターと異なる、あるいは、それを超えてなされた説明で当該TI製品もしくはサービスを再販売することは、当該TI製品もしくはサービスに対する全ての明示的保証、及び何らかの黙示的保証を無効にし、かつ不公正で誤認を生じさせる行為です。TIは、そのような説明については何の義務も責任もありません。

TIは、TIの製品が、安全でないことが致命的となる用途ないしアプリケーション（例えば、生命維持装置のように、TI製品に不良があった場合に、その不良により相当な確率で死傷等の重篤な事故が発生するようなもの）に使用されることを認めておりません。但し、お客様とTIの双方の権限有る役員が書面でそのような使用について明確に合意した場合は除きます。たとえTIがアプリケーションに関連した情報やサポートを提供したとしても、お客様は、そのようなアプリケーションの安全面及び規制面から見た諸問題を解決するために必要とされる専門的知識及び技術を持ち、かつ、お客様の製品について、またTI製品をそのような安全でないことが致命的となる用途に使用することについて、お客様が全ての法的責任、規制を遵守する責任、及び安全に関する要求事項を満足させる責任を負っていることを認め、かつそのことに同意します。さらに、もし万一、TIの製品がそのような安全でないことが致命的となる用途に使用されたことによって損害が発生し、TIないしその代表者がその損害を賠償した場合は、お客様がTIないしその代表者にその全額の補償をするものとします。

TI製品は、軍事的用途もしくは宇宙航空アプリケーションないし軍事的環境、航空宇宙環境にて使用されるようには設計もされていませんし、使用されることを意図されておりません。但し、当該TI製品が、軍需対応グレード品、若しくは「強化プラスティック」製品としてTIが特別に指定した製品である場合は除きます。TIが軍需対応グレード品として指定した製品のみが軍需品の仕様書に合致いたします。お客様は、TIが軍需対応グレード品として指定していない製品を、軍事的用途もしくは軍事的環境下で使用することは、もっぱらお客様の危険負担においてなされるということ、及び、お客様がもっぱら責任をもって、そのような使用に関して必要とされる全ての法的要求事項及び規制上の要求事項を満足させなければならないことを認め、かつ同意します。

TI製品は、自動車用アプリケーションないし自動車の環境において使用されるようには設計されていませんし、また使用されることを意図されておりません。但し、TIがISO/TS 16949の要求事項を満たしていると特別に指定したTI製品は除きます。お客様は、お客様が当該TI指定品以外のTI製品を自動車用アプリケーションに使用しても、TIは当該要求事項を満たしていなかったことについて、いかなる責任も負わないことを認め、かつ同意します。

弊社半導体製品の取り扱い・保管について

半導体製品は、取り扱い、保管・輸送環境、基板実装条件によっては、お客様での実装前後に破壊/劣化、または故障を起こすことがあります。

弊社半導体製品のお取り扱い、ご使用にあたっては下記の点を遵守して下さい。

1. 静電気
   - 素手で半導体製品単体を触らないこと。どうしても触る必要がある場合は、リストストラップ等で人体からアースをとり、導電性手袋等をして取り扱うこと。
   - 弊社出荷梱包単位（外装から取り出された内装及び個装）又は製品単品で取り扱いを行う場合は、接地された導電性のテーブル上で（導電性マットにアースをとったもの等）、アースをした作業者が行うこと。また、コンテナ等も、導電性のものを使うこと。
   - マウンタやはんだ付け設備等、半導体の実装に関わる全ての装置類は、静電気の帯電を防止する措置を施すこと。
   - 前記のリストストラップ・導電性手袋・テーブル表面及び実装装置類の接地等の静電気帯電防止措置は、常に管理されその機能が確認されていること。
2. 温・湿度環境
   - 温度：0～40℃、相対湿度：40～85%で保管・輸送及び取り扱いを行うこと。（但し、結露しないこと。）
   - 直射日光があたる状態で保管・輸送しないこと。
3. 防湿梱包
   - 防湿梱包品は、開封後は個別推奨保管環境及び期間に従い基板実装すること。
4. 機械的衝撃
   - 梱包品（外装、内装、個装）及び製品単品を落下させたり、衝撃を与えないこと。
5. 熱衝撃
   - はんだ付け時は、最低限260℃以上の高温状態に、10秒以上さらさないこと。（個別推奨条件がある時はそれに従うこと。）
6. 汚染
   - はんだ付け性を損なう、又はアルミ配線腐食の原因となるような汚染物質（硫黄、塩素等ハロゲン）のある環境で保管・輸送しないこと。
   - はんだ付け後は十分にフラックスの洗浄を行うこと。（不純物含有率が一定以下に保証された無洗浄タイプのフラックスは除く。）

以上