TEXAS INSTRUMENTS

参 考 資 料

TMS320 DSP/BIOS

ユーザーズ・ガイド

2006

Digital Signal Processing Solutions

SPRU423 翻訳版

JAJU022 WAS SPRU570A

TMS320 DSP/BIOS
ユーザーズ・ガイド

対応英文マニュアル：SPRU423F
2004 年 11 月

2006 年 9 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

まえがき

最初にお読みください

本書について

DSP/BIOS を使用すると、TI の TMS320 DSP デバイス上のメインストリーム・アプリケーションの開発者は、リアルタイムの組み込み型ソフトウェアを開発することができます。DSP/BIOS は、ファームウェア用の小規模なリアルタイムのライブラリ、およびリアルタイム・トレースやリアルタイム解析のための使いやすいツールを提供します。

ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』を読んで、よく理解することを推奨します。『API Reference Guide』は、本書と一対の文献です。

また、本書をお読みになる前に、オンライン・ヘルプの『Code Composer Studio Tutorial』にある「DSP/BIOS の使用方法」に従って、DSP/BIOS の概要を理解しておくことを推奨します。本書では、DSP/BIOS について種々の局面から詳細に解説しています。また、DSP/BIOS のその他の局面についても、少なくとも基本的なことは理解していることを想定しています。

『TMS320 DSP/BIOS User's Guide』（文献番号 SPRU423F）を翻訳しています。

表記規則

本書では、次の表記規則を使用しています。

- プログラム・リスト、プログラム例、および対話表示は、特殊な活字（special typeface）で示してあります。例は強調のため、ボールド（**bold version**）で示してあります。対話表示についても、ユーザが入力するコマンドとシステムが表示する項目（プロンプト、コマンド出力、エラー・メッセージなど）とを区別するために、ボールド（**bold version**）で示してあります。

  プログラム・リストの例を次に示します。

```
Void copy(HST_Obj *input, HST_Obj *output)
{
    PIP_Obj     *in, *out;
    Uns         *src, *dst;
    Uns         size;
}
```

- ❑ 大括弧（[ ]）は、任意のパラメータを示します。任意のパラメータを使用する場合、この括弧内の情報を入力します。大括弧が**ボールド**である場合を除き、大括弧そのものは入力不要です。

- ❑ 本書では、54 は、特定の DSP プラットフォームに適切な 2 桁の数字を表しています。ご使用の DSP プラットフォームが C62x ベースの場合は、本書に出てくる 54 はいつでも 62 に読み替えてください。たとえば、C6000 プラットフォーム用の DSP/BIOS アセンブリ言語の API ヘッダ・ファイルのサフィックスは .h62 です。C2800 プラットフォームのサフィックスは .h28 になります。C64x、C55x、C28x DSP プラットフォームの場合は、54 と書かれていたら 64、55、28 にそれぞれ読み替えてください。また、Code Composer Studio C5000 への各参照は、ご使用の DSP プラットフォームに応じて Code Composer Studio C6000 や Code Composer Studio C2000 に読み替えてください。

- ❑ 特定のデバイスに固有の情報は、次のいずれかのアイコンで示されます。

当社発行の関連文献

TMS320 デバイスおよびそのサポート・ツールを解説した関連文献は次のとおりです。当社の刊行物を入手するには、TI の Web サイト www.ti.com をご覧ください。

TMS320C28x DSP/BIOS API Reference Guide（文献番号 SPRU625）
TMS320C5000 DSP/BIOS API Reference Guide（文献番号 SPRU404）
TMS320C6000 DSP/BIOS API Reference Guide（文献番号 SPRU403）は、DSP/BIOS API の関数についてアルファベット順に解説しています。『API Reference Guide』は、本書と一対の文献です。

DSP/BIOS Textual Configuration (Tconf) User's Guide（文献番号 SPRU007）は、DSP/BIOS アプリケーションを構成するために使用するスクリプト言語について解説しています。

DSP/BIOS Driver Developer's Guide（文献番号 SPRU616）は、デバイス・ドライバを開発し、DSP/BIOS アプリケーションに統合するための IOM モデルを解説しています。

Code Composer Studio Online Help には、Code Composer Studio の操作方法が記述されています。「DSP/BIOS」セクションには、本書では説明していない DSP/BIOS を使用する手順が順を追って記述されています。

Code Composer Studio Online Tutorial は、Code Composer Studio の統合開発環境およびソフトウェア・ツールを紹介しています。「Using DSP/BIOS 」レッスンは、DSP/BIOS ユーザにとって特に興味深いものになっています。

TMS320C2000 Assembly Language Tools User's Guide（文献番号 SPRU513）
TMS320C54x アセンブリ言語ツール ユーザーズ・マニュアル（文献番号 SPRU366）
TMS320C55x アセンブリ言語ツール ユーザーズ・マニュアル（文献番号 SPRUE46）
TMS320C6000 Assembly Language Tools User's Guide（文献番号 SPRU186）は、C5000 世代のデバイス用のアセンブリ言語ツール（アセンブラ、リンカ、およびその他のアセンブリ言語コード開発ツール）、アセンブラ疑似命令、マクロ、共通オブジェクト・ファイル・フォーマット、およびシンボリック・デバッグ疑似命令について解説しています。

TMS320C2000 Optimizing C/C++ Compiler User's Guide（文献番号 SPRU514）は、C2000 C/C++ コンパイラおよびアセンブリ・オプティマイザについて解説しています。この C/C++ コンパイラは、ANSI 準拠の標準 C/C++ ソース・コードを受け入れ、C2000 世代のデバイス用のアセンブリ言語ソース・コードを作成します。アセンブリ・オプティマイザにより、アセンブリ・コードを最適化することができます。

TMS320C54x オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル（文献番号 SPRU367）は、C54x C コンパイラについて解説しています。この C/C++ コンパイラは、ANSI 準拠の標準 C/C++ ソース・コードを受け入れ、C54x 世代のデバイス用の TMS320 アセンブリ言語ソース・コードを作成します。

TMS320C55x オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル（文献番号 SPRUE45）は、C55x C コンパイラについて解説しています。この C/C++ コンパイラは、ANSI 準拠の標準 C/C++ ソース・コードを受け入れ、C55x 世代のデバイス用の TMS320 アセンブリ言語ソース・コードを作成します。

TMS320C6000 オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル（文献番号 SPRU419）は、C6000 C/C++ コンパイラおよびアセンブリ・オプティマイザについて解説しています。この C/C++ コンパイラは、ANSI 準拠の標準 C/C++ ソース・コードを受け入れ、C6000 世代のデバイス用のアセンブリ言語ソース・コードを作成します。アセンブリ・オプティマイザにより、アセンブリ・コードを最適化することができます。

TMS320C55x Programmer's Guide（文献番号 SPRU376）は、TMS320C55x DSP 用に C およびアセンブリ・コードを最適化する方法について解説しています。また、アプリケーション・プログラム例も記載されています。

TMS320C6000 Programmer's Guide（文献番号 SPRU189）は、C6000 デジタル・シグナル・プロセッサの CPU アーキテクチャ、命令セット、パイプライン、および割り込みについて解説しています。

TMS320C54x DSP リファレンス・セット、Volume 1: CPU 及びペリフェラル（文献番号 SCJ2287）は、TMS320C54x 16 ビット固定小数点汎用デジタル・シグナル・プロセッサについて解説しています。このプロセッサのアーキテクチャ、内部レジスタ構成、データおよびプログラムのアドレッシング、命令パイプライン、およびオンチップ・ペリフェラルについて記載しています。また、開発サポート情報、パーツ・リスト、および XDS510 エミュレータを使用する際のデザイン上の考慮事項についても解説しています。

TMS320C54x DSP リファレンス・セット、Volume 5: 高機能ペリフェラル（文献番号 SPRU396）は、TMS320C54x デジタル・シグナル・プロセッサで使用可能な高機能ペリフェラルについて解説しています。マルチチャネル・バッファド・シリアル・ポート（McBSP）、ダイレクト・メモリ・アクセス（DMA）コントローラ、プロセッサ間通信、HPI-8 および HPI-16 ホスト・ポート・インターフェイスについても解説しています。

TMS320C54x DSP Mnemonic Instruction Set Reference Set Volume 2（文献番号 SPRU172）は、TMS320C54x デジタル・シグナル・プロセッサのニーモニック命令について個別に解説しています。また、命令セットのクラスとサイクルの一覧も含まれています。

TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set（文献番号 SPRU179）は、TMS320C54x デジタル・シグナル・プロセッサは代数表記命令について個別に解説しています。また、命令セットのクラスとサイクルの一覧も含まれています。

TMS320C6000 Peripherals Reference Guide（文献番号 SPRU190）は、TMS320C6000 ファミリーのデジタル・シグナル・プロセッサで使用可能な共通ペリフェラルについて解説しています。内部データ・メモリおよびプログラム・メモリ、外部メモリ・インターフェイス（EMIF）、ホスト・ポート、マルチチャネル・バッファド・シリアル・ポート、ダイレクト・メモリ・アクセス（DMA）、クロック・フェーズ・ロックド・ループ（PLL）、およびパワーダウン・モードに関する情報も含まれています。

DSP/BIOS and TMS320C54x Extended Addressing（文献番号 SPRA599）は、アプリケーション計測に対応したリアルタイム解析関数、クロックおよび周期関数、入出力（I/O）モジュール、および優先スケジューラなどの基本的な実行サービスについて解説しています。また、TMS320C54x プラットフォームで使用可能な拡張アドレッシング用の far モデルについても解説しています。

TMS320C28x DSP and CPU Instruction Reference Guide（文献番号 SPRU430）。

関連文献

本書の補足資料は、次のとおりです。

The C Programming Language（第 2 版）－ Brian W. Kernighan と Dennis M. Ritchie の共著、Prentice Hall、Englewood Cliffs（New Jersey）発行（1988）

Programming in C － Kochan, Steve G. 著、Hayden Book Company 発行

Programming Embedded Systems in C and C++ － Michael Barr 著、Andy Oram 編集、O'Reilly & Associates 発行、ISBN: 1565923545（February 1999）

Real-Time Systems － Jane W. S. Liu 著、Prentice Hall 発行、ISBN: 013099651（June 2000）

Principles of Concurrent and Distributed Programming（Prentice Hall International Series in Computer Science）－ M. Ben-Ari 著、Prentice Hall 発行、ISBN: 013711821X（May 1990）

American National Standard for Information Systems-Programming Language C －米国規格協会発行（C 言語に関する ANSI 規格）（絶版）

商標

MS-DOS、Windows および Windows NT は、Microsoft Corporation の商標です。

Texas Instruments のロゴ、および Texas Instruments は、Texas Instruments の登録商標です。Texas Instruments の商標には、TI、XDS、Code Composer、Code Composer Studio、Probe Point、Code Explorer、DSP/BIOS、RTDX、Online DSP Lab、BIOSuite、SPOX、TMS320、TMS320C54x、TMS320C55x、TMS320C62x、TMS320C64x、TMS320C67x、TMS320C28x、TMS320C5000、TMS320C6000 および TMS320C2000 が含まれています。

その他のすべてのブランド名または製品名は、それぞれの会社または団体の商標あるいは登録商標です。

目次

図

表

例

第 1 章

DSP/BIOS について

DSP/BIOS は、スケーラブルなリアルタイム・カーネルです。このカーネルは、リアルタイムのスケジューリングと同期、ホストとターゲット間の通信、またはリアルタイムの計測を必要とするアプリケーションを対象として、設計されています。DSP/BIOS は、プリエンプティブなマルチスレッド機能、ハードウェア抽象機能、リアルタイム解析、および構成ツールを提供します。

項目 **ページ**

1.1 DSP/BIOS の特徴と利点

DSP/BIOS は、ターゲットで必要とされるメモリ量と CPU 要件が最小限で済むように設計されています。この設計目標を達成するために、次の方法を採用しています。

- ❑ すべての DSP/BIOS オブジェクトを静的に構成し、実行可能なプログラム・イメージにバインドすることができます。これにより、コード・サイズは小さくなり、内部データ構造が最適化されます。
- ❑ 計測データ（ログやトレースなど）は、ホスト上で書式化されます。
- ❑ API はモジュール設計になっているので、プログラムで使用される API の部分だけを実行可能なプログラムにバインドするだけで済みます。
- ❑ ライブラリは、大半がアセンブリ言語で記述されていて、最小限の命令サイクル数で実行できるように最適化されています。
- ❑ ターゲットと DSP/BIOS 解析ツール間の通信は、バックグラウンドのアイドル・ループの間に実行されます。したがって、DSP/BIOS 解析ツールがプログラム・タスクの実行を妨げることはありません。ターゲット CPU が使用中でバックグラウンド・タスクを実行できない場合、DSP/BIOS 解析ツールは、CPU が使用可能になるまでターゲットからの情報の受信を停止します。
- ❑ メモリ量と CPU 要件を増やすエラー・チェック機能は、必要最小限に抑えられています。また、API リファレンスでは、API 関数を呼び出す際の制約事項を示しています。アプリケーション開発者は、これらの制約事項を守る責任があります。

さらに、DSP/BIOS API は、プログラムを開発する際の多数のオプションを提供します。

- ❑ プログラムは、特殊な状況下で使用するオブジェクトを動的に作成したり削除したりすることができます。同一プログラムで、動的に作成されたオブジェクトと静的に作成されたオブジェクトの両方を使用することができます。
- ❑ スレッド化モデルは、さまざまな状況に対応するスレッド・タイプを提供します。ハードウェア割り込み、ソフトウェア割り込み、タスク、アイドル関数、および周期関数のすべてがサポートされています。開発者は、スレッド・タイプを選択することにより、スレッドの優先順位とブロック化特性を制御することができます。
- ❑ スレッド間の通信と同期をサポートする構造体が提供されています。これらの構造体には、セマフォ、メールボックス、リソース・ロックなどがあります。
- ❑ 2 つの入出力（I/O）モデルがサポートされているため、最大限の柔軟性と能力を発揮します。パイプはターゲット / ホスト間通信に利用され、1 つのスレッドがパイプに書き込み、別のスレッドがパイプから読み取るといった単純な状況をサポートするために使用します。ストリームは、さらに複雑な入出力（I/O）用としてデバイス・ドライバをサポートするために使用されます。
- ❑ エラーの処理、共通データ構造体の作成、およびメモリの使用状況を管理しやすくするために、低レベルのシステム・プリミティブが提供されます。

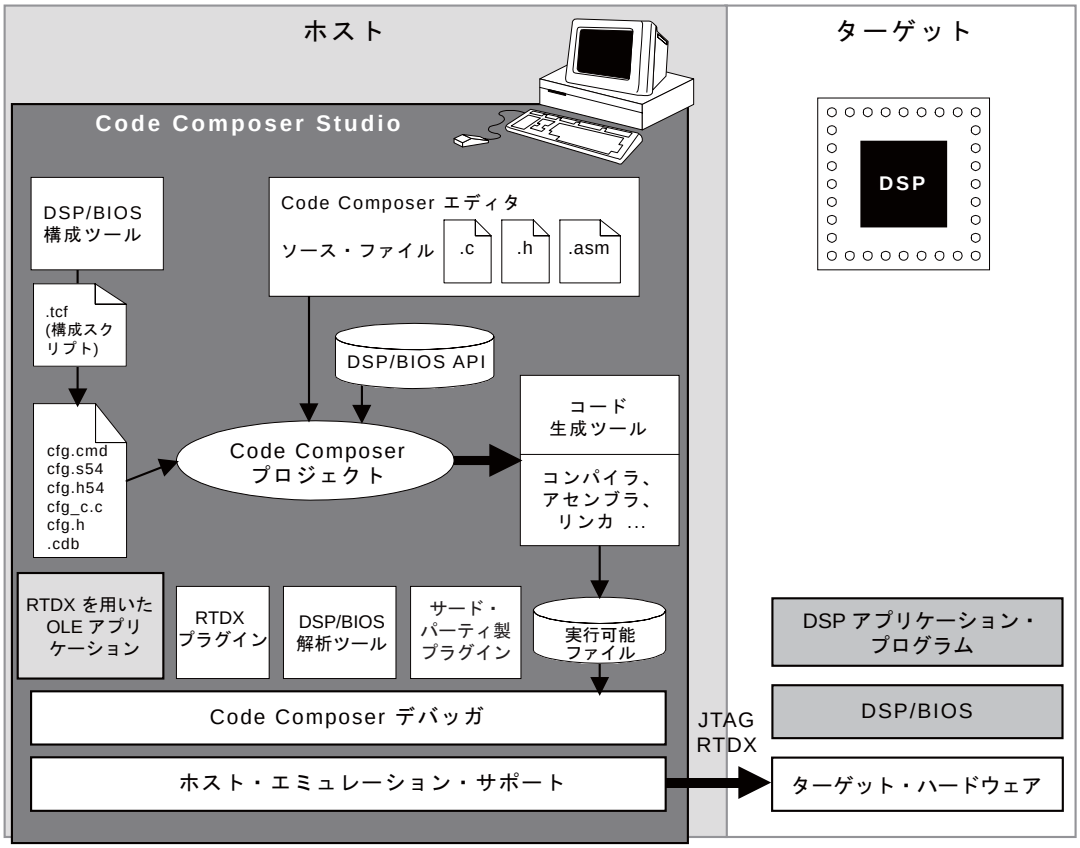
DSP/BIOS API は、多くの TI デバイスに対応した DSP プログラミングを標準化します。また、使いやすくパワフルなプログラム開発ツールを提供します。これらのツールにより、次に示すように、DSP プログラムを作成するために必要な時間が短縮されます。

- ❏ Tconf 構成スクリプトは、プログラム内で使用されるオブジェクトを静的に宣言するために必要なコードを生成します。
- ❏ 構成ツールは、プログラムがビルドされる前にプロパティを検証することで、より素早くエラーを検出します。
- ❏ 分岐、ループ、コマンドライン引数のテストなどを含めるために、テキスト・エディタで構成スクリプトを変更することができます。
- ❏ DSP/BIOS オブジェクトのロギングや統計情報は追加のプログラミングをしなくても実行時に使用することができます。必要に応じて、追加の計測処理をプログラミングすることもできます。
- ❏ DSP/BIOS 解析ツール により、プログラムの動作をリアルタイムで監視することができます。
- ❏ DSP/BIOS は標準 API を提供します。この API を使用することで、DSP アルゴリズムの開発担当者は、他のプログラム機能に簡単に統合できるコードを作成することができます。
- ❏ DSP/BIOS は Code Composer Studio IDE に統合されています。ランタイム・ライセンス料金は不要で、しかも当社が完全なサポートを提供しています。DSP/BIOS は、TI の eXpressDSP™ リアルタイム・ソフトウェア・テクノロジーのキー・コンポーネントの 1 つです。

1.2 DSP/BIOS のコンポーネント

図 1-1 に、Code Composer Studio のプログラム生成およびデバッグ環境で使用する DSP/BIOS のコンポーネントを示します。

図 1-1. DSP/BIOS のコンポーネント



- **DSP/BIOS API:** ホスト PC 上で、DSP/BIOS API 関数を呼び出すプログラムを、（C、C++、またはアセンブリ言語を使用して）記述します。
- **DSP/BIOS 構成ツール :** 構成スクリプトを作成して、プログラム内で使用する静的なオブジェクトを定義します。構成スクリプトでは、プログラムをコンパイルし、リンクするファイルを指定します。
- **DSP/BIOS 解析ツール :** Code Composer Studio の DSP/BIOS 解析ツールを使用して、ターゲット・デバイス上でプログラムをテストし、CPU の負荷、タイミング、ログ、スレッドの実行などを監視します（「スレッド」という用語は、すべての実行スレッド、つまりハードウェア割り込み、ソフトウェア割り込み、タスク、アイドル関数を表します）。

ここでは、DSP/BIOS の各種コンポーネントの概要を説明します。

1.2.1 DSP/BIOS のリアルタイム・カーネルと API

DSP/BIOS はスケーラブルなリアルタイム・カーネルで、リアルタイム・スケジューリングと同期、ホストとターゲット間の通信、またはリアルタイム計測を必要とするアプリケーションを対象に設計されています。DSP/BIOS は、プリエンプティブなマルチスレッド機能、ハードウェア抽象機能、リアルタイム解析、および構成ツールを提供します。

DSP/BIOS API は、複数のモジュールに分かれています。構成されるモジュール、およびアプリケーションが使用するモジュールに基づいて、DSP/BIOS のコード・サイズは、約 500 から 6500 ワードの範囲になります。モジュール内のすべてのオペレーションは、表 1-1 に示す文字コードで始まります。

アプリケーション・プログラムは、API を呼び出すことで DSP/BIOS を使用します。DSP/BIOS のすべてのモジュールは、C 呼び出し可能なインターフェイスを提供しています。ほとんどの C 呼び出し可能なインターフェイスは、C の呼び出し規約に従っていれば、アセンブリ言語からも呼び出すことができます。一部の C インターフェイスは実際には C マクロであり、したがって、アセンブリ言語から呼び出して使用することはできません。詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』を参照してください。

表 1-1. DSP/BIOS のモジュール

モジュール	説明
ATM	アセンブリ言語で記述された極小の関数
BUF	固定長のバッファ・プール・マネージャ
C28、C54、C55、C62、C64	ターゲット固有の関数、プラットフォーム依存
CLK	クロック・マネージャ
DEV	デバイス・ドライバ・インターフェイス
GBL	グローバル設定マネージャ
GIO	汎用入出力（I/O）マネージャ
HOOK	フック関数マネージャ
HST	ホスト・チャネル・マネージャ
HWI	ハードウェア割り込みマネージャ
IDL	アイドル関数マネージャ
LCK	リソース・ロック・マネージャ
LOG	イベント・ログ・マネージャ
MBX	メールボックス・マネージャ
MEM	メモリ・セグメント・マネージャ
MSGQ	メッセージ・キュー・マネージャ
PIP	バッファド・パイプ・マネージャ
POOL	アロケータ・プール・マネージャ

表 1-1. DSP/BIOS のモジュール（続き）

モジュール	説明
PRD	周期関数マネージャ
PWRM	電源マネージャ（C55x のみ）
QUE	極小キュー・マネージャ
RTDX	リアルタイム・データ・エクスチェンジ設定
SEM	セマフォ・マネージャ
SIO	ストリーム入出力（I/O）マネージャ
STS	統計オブジェクト・マネージャ
SWI	ソフトウェア割り込みマネージャ
SYS	システム・サービス・マネージャ
TRC	トレース・マネージャ
TSK	マルチタスキング・マネージャ

1.2.2 DSP/BIOS 構成

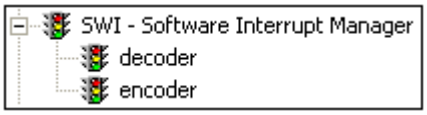
DSP/BIOS 構成を使用すると、実行時ではなく静的にオブジェクトを作成し、そのプロパティを設定できるので、アプリケーションを最適化することができます。これにより、実行時性能を改善し、アプリケーションの占有スペースを削減します。

構成のソース・ファイルは、.tcf というファイル拡張子をもつ DSP/BIOS Tconf のスクリプトです。DSP/BIOS 構成にアクセスするには、次の 2 つの方法があります。

- **テキストを介して。**スクリプトのテキストは、Code Composer Studio または別のテキスト・エディタを使用して編集することができます。JavaScript 構文を使用して、構成を記述します。詳細については、『DSP/BIOS Textual Configuration (Tconf) User's Guide』（文献番号 SPRU007）を参照してください。

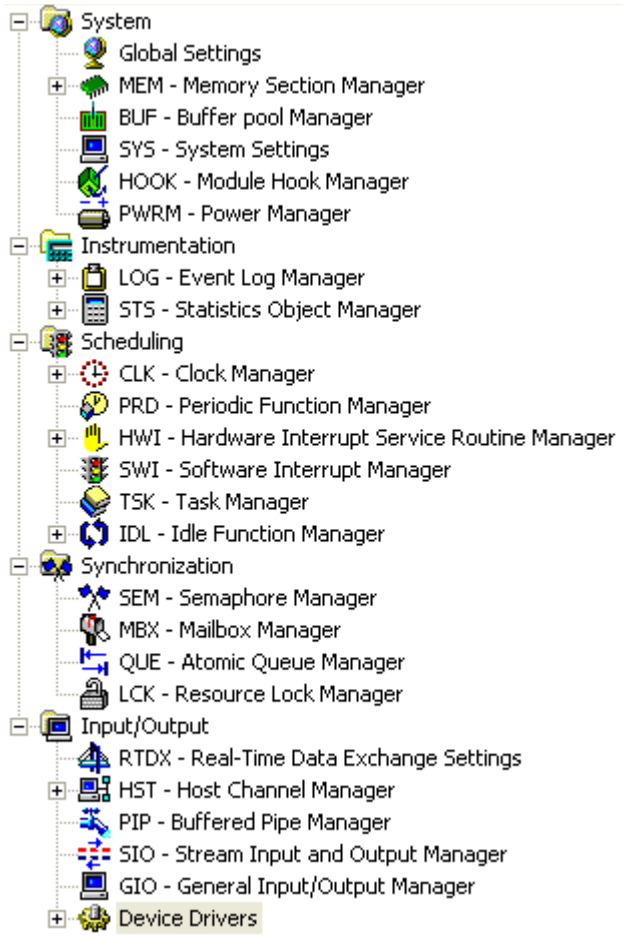
```
prog.module("SWI").create("encoder");
prog.module("SWI").create("decoder");
```

- **GUI を介して。**構成スクリプトは、DSP/BIOS 構成ツールを使って読み取り専用モードで表示することができます。構成ツールのインターフェイスは、Windows のエクスプローラとよく似ています。



DSP/BIOS が実行時に使用する、さまざまなパラメータをセットすることができます。作成するオブジェクトは、アプリケーションの DSP/BIOS API コールを用いて使用します。これらのオブジェクトには、ソフトウェア割り込み、タスク、入出力（I/O）ストリーム、イベント・ログなどが含まれます。

図 1-2.　構成ツールのモジュール階層



構成を保存すると、Tconf はプロジェクトに含まれるファイルを生成します。静的な構成を使用して、DSP/BIOS オブジェクトを事前に構成し、実行可能なプログラム・イメージにバインドすることができます。また、DSP/BIOS プログラムは、特定のオブジェクトの作成や削除を実行時に行うこともできます。

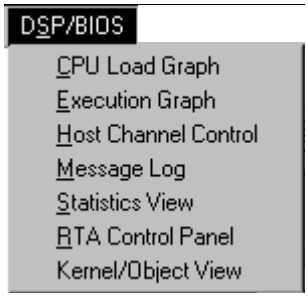
静的にオブジェクトを作成すると、ランタイム・コードが排除され、内部データ構造が最適化されるために、ターゲット・メモリの必要量を最小限に抑えることができるだけでなく、プログラムをコンパイルする前にオブジェクトのプロパティを確認できるので、早期にエラーを検出することができます。

詳細については、DSP/BIOS のオンライン・ヘルプや 2.2 節「DSP/BIOS アプリケーションを静的に構成する方法」（2-3 ページ）を参照してください。

1.2.3 DSP/BIOS 解析ツール

DSP/BIOS 解析ツールは、Code Composer Studio 環境の補完機能として働き、DSP/BIOS アプリケーション・プログラムのリアルタイム解析を可能にします。これを使用すると、アプリケーションのリアルタイム・パフォーマンスへの影響を最小限に抑えて、実行中の DSP アプリケーションを視覚的にモニタすることができます。DSP/BIOS 解析ツールは、図 1-3 に示すように DSP/BIOS メニューに表示されます。

図 1-3. Code Composer Studio の DSP/BIOS メニュー



個々の解析ツールの詳細については、DSP/BIOS のオンライン・ヘルプや 第 3 章「計測」を参照してください。

従来型のデバッグが実行中のプログラムの外部で行われるのに対して、プログラム解析では、ターゲット・プログラムにリアルタイムの計測サービスが含まれている必要があります。DSP/BIOS API およびオブジェクトを使用すると、開発者は Code Composer Studio の DSP/BIOS 解析ツールを経由してターゲットから情報をリアルタイムに取り込んでホストにアップロードする自動計測環境を容易に実現することができます。

次のようなリアルタイムのプログラム解析機能が提供されています。

- **プログラム・トレース**： ターゲット・ログに書き込まれたイベントを表示します。プログラム実行中の動的な制御フローを表します。
- **パフォーマンス・モニタ**： 統計情報を表示します。これには、ターゲット・リソースの使用状況（プロセッサの負荷やタイミングなど）を表します。
- **ファイル・ストリーム**： ターゲット常駐の入出力（I/O）オブジェクトを、ホスト・ファイルにバインドします。

DSP/BIOS のリアルタイム解析ツールを Code Composer Studio の他のデバッグ機能と併せて使用すれば、ターゲット・プログラムの実行中にプログラムの動作を事細かに調べることができます（ターゲットを停止して行う従来のデバッグ技法では、厳密な情報はほとんど得られません）。デバッガがプログラムを停止した後でも、ホストが DSP/BIOS 解析ツールを使用してすでに取り込んだ情報から、現在の実行時点に至るまでのイベントのシーケンスを正確に把握することができます。

ソフトウェア開発サイクルの進行にともない、時間依存型の相互作用が原因で通常のデバッグ技法では対処できないような問題が生じたとき、DSP/BIOS 解析ツールはハードウェアのロジック・アナライザを補完するソフトウェア機能として、さらに重要な役割を果たします。

図 1-4 に、さまざまな DSP/BIOS 解析ツールを示します。

図 1-4.　Code Composer Studio 解析ツールのパネル

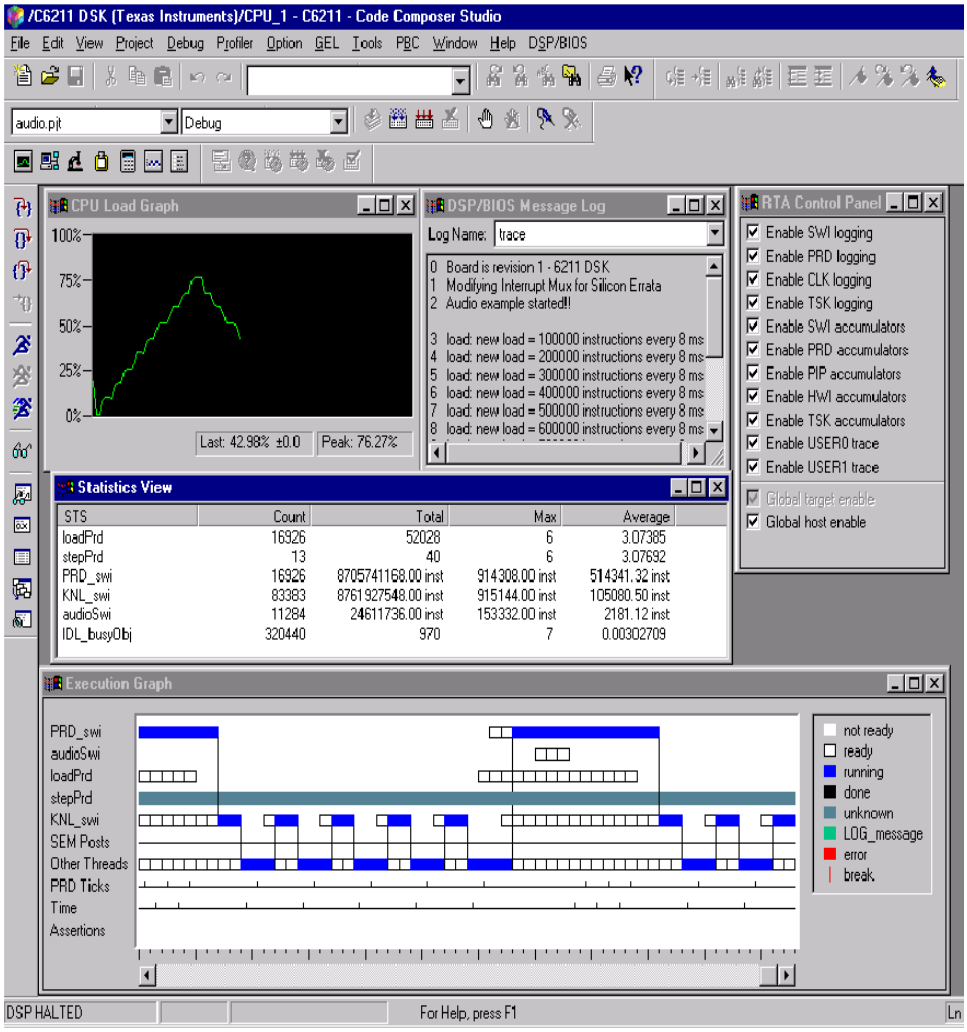
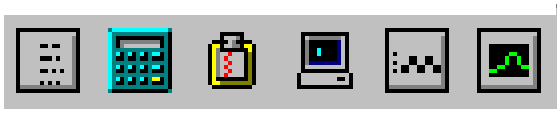


図 1-5 に、DSP/BIOS 解析ツールのツールバーを示します。このツールバーのオンまたはオフを切り替えるには、[View]→[Plug-in Toolbar]→[DSP/BIOS] の順に選択します。

図 1-5.　DSP/BIOS 解析ツールのツールバー



1.3 命名規約

各 DSP/BIOS の各モジュールには、そのモジュールの演算（関数）、ヘッダ・ファイル、およびオブジェクトのプリフィックスとして使用される固有名が付いています。この名前は、3 文字以上の大文字の英数字から構成されています。

本書では、54 は、ご使用の DSP プラットフォームに該当する 2 桁の数字を表すものとします。ご使用の DSP プラットフォームが C6200 ベースの場合、54 という表記が書かれていたら 62 と読み替えてください。たとえば、C6000 プラットフォーム用の DSP/BIOS アセンブリ言語の API ヘッダ・ファイルは、拡張子が .h62 になります。C5000 DSP プラットフォームの場合、54 と書かれていたら 54 または 55 のいずれかに読み替えてください。また、Code Composer Studio C5000 と示されている個所は、必要に応じて Code Composer Studio C6000 と読み替えてください。

大文字で始まりその後に下線が続く識別子（XXX_*）は、すべて予約語と見なされます。

1.3.1 モジュール・ヘッダ名

DSP/BIOS の各モジュールにはそれぞれ 2 つのヘッダ・ファイルがあり、そのモジュールのインターフェイスを介して使用できるすべての定数、型、および関数宣言が含まれています。

- **xxx.h** : C プログラムの DSP/BIOS API ヘッダ・ファイル。C ソース・ファイルには、std.h ファイル、および C 関数で使用するすべてのモジュール用のヘッダ・ファイルをインクルードする必要があります。
- **xxx.h54** : アセンブリ・プログラムの DSP/BIOS API ヘッダ・ファイル。アセンブリ・ソース・ファイルには、アセンブリ・ソースで使用するすべてのモジュール用の xxx.h54 ヘッダ・ファイルをインクルードする必要があります。このファイルには、このデバイスに固有のマクロ定義が入っています。

プログラムには、特定のプログラム・ソース・ファイル内で使用される各モジュールに対応するヘッダをインクルードする必要があります。さらに、C ソース・ファイルでは、他のモジュール・ヘッダ・ファイルより前に std.h をインクルードする必要があります（詳細については、1.3.4 節「データ型の名前」（1-12 ページ）を参照）。std.h ファイルには、標準の型および定数の定義が含まれています。std.h をインクルードした後は、他のヘッダ・ファイルを任意の順序でインクルードすることができます。次に例を示します。

```
#include <std.h>
#include <tsk.h>
#include <sem.h>
#include <prd.h>
#include <swi.h>
```

DSP/BIOS には、内部で使用する多数のモジュールが含まれています。これらの内部モジュールに関する解説資料はありません。また、これらのモジュールは随時変更されることがあります。これらの内部モジュール用のヘッダ・ファイルは DSP/BIOS の一部として配布されるものであり、DSP/BIOS プログラムをコンパイルおよびリンクするときには、必ずシステムに存在していなければなりません。

1.3.2 オブジェクト名

デフォルトでは構成に組み込まれるシステム・オブジェクトには、一般に、そのオブジェクトを定義または使用するモジュールを表す 3 ～ 4 文字のコードで始まる名前が付いています。たとえば、デフォルトの構成には、LOG_system という名前の LOG オブジェクトが含まれています。

注 :

静的に作成するオブジェクトには、統一した命名規則を適用する必要があります。たとえば、オブジェクト名のサフィックスとしてモジュール名を使用する方法が考えられます。その場合、データをエンコードする TSK オブジェクトがあるとすれば、その名前は encoderTsk になります。

1.3.3 オペレーション名

DSP/BIOS API オペレーション名の形式は MOD_action です。ここで、MOD は該当のオペレーションが含まれているモジュールを表す文字コードで、action はそのオペレーションにより実行されるアクションです。たとえば、SWI_post 関数は SWI モジュールにより定義されます。この関数は、ソフトウェア割り込みを通知します。

本製品に実装されている DSP/BIOS API にも、さまざまなビルトイン・オブジェクトで実行できるいくつかのビルトイン関数が含まれています。次に、その例をいくつか示します。

- **CLK_F_isr** : HWI オブジェクトにより実行され、低分解能の CLK ティックを提供します。
- **PRD_F_tick** : CLK オブジェクト PRD_clock により実行され、PRD_SWI およびシステム・ティックを管理します。
- **PRD_F_swi** : PRD_tick によりトリガされ、PRD 関数を実行します。
- **_KNL_run** : 優先順位が最も低い SWI オブジェクト KNL_swi により実行され、タスク・スケジューラがイネーブルの場合には、それを実行します。これは KNL_run という名前の C 関数です。プリフィックスとして下線が付いているのは、アセンブリ・コードからこの関数を呼び出すからです。
- **_IDL_loop** : 優先順位が最も低い TSK オブジェクト TSK_idle により実行され、IDL 関数を実行します。
- **IDL_F_busy** : IDL オブジェクト IDL_cpuLoad により実行され、現在の CPU の負荷を計算します。
- **RTA_F_dispatch** : IDL オブジェクト RTA_dispatcher により実行され、リアルタイム解析データを収集します。

- **LNK_F_dataPump**：IDL オブジェクト LNK_dataPump により実行され、ホストへのリアルタイム解析データおよび HST チャネル・データの転送を管理します。
- **HWI_unused**：これは実際には関数名ではありません。この文字列は構成スクリプトの中で使用され、未使用の HWI オブジェクトにマーク付けします。

注：

ユーザのプログラム・コードでは、MOD_F_ で始まる名前のビルトイン関数は呼び出さないでください。このような関数は、構成時に指定した関数パラメータとしてのみ呼び出す目的で設計されています。

MOD_ および MOD_F_（ここで MOD は DSP/BIOS モジュールを表す文字コード）で始まるシンボル名は、内部で使用するために予約されています。

1.3.4 データ型の名前

DSP/BIOS API は、C の基本型（int や char など）を明示的に使用しません。代わりに、DSP/BIOS API をサポートする他のプロセッサへの移植性を確保するために、DSP/BIOS では独自の標準データ型を定義しています。ほとんどの場合、DSP/BIOS の標準型は、対応する C の型を大文字にしたものです。

表 1-2 に示すデータ型は、std.h ヘッダ・ファイルの中で定義されているものです。

表 1-2. DSP/BIOS の標準データ型

型	説明
Arg	Ptr および Int の両方の引数を保持できる型
Bool	ブール値
Char	文字値
Fxn	関数へのポインタ
Int	符号付き整数値
LgInt	長精度符号付き整数値
LgUns	長精度符号なし整数値
Ptr	汎用ポインタ値
String	ゼロで終了する（\0）文字シーケンス（配列）
Uns	符号なし整数値
Void	空の型

std.h の中では、他のデータ型が定義されていますが、DSP/BIOS API では使用されません。

さらに DSP/BIOS では、空のポインタ値を表すために、標準定数 NULL (0) が使用されます。定数 TRUE (1) および FALSE (0) は、Bool 型の値に使用されます。

DSP/BIOS API モジュールが使用するオブジェクト構造には、MOD_Obj という命名規則が適用されます。ここで MOD は、オブジェクトのモジュールを表す文字コードです。構成時に作成したこのようなオブジェクトをプログラム・コード内で使用する場合は、そのオブジェクトを extern 宣言する必要があります。次に例を示します。

```
extern LOG_Obj trace;
```

構成スクリプトを実行すると、自動的に C ヘッダを生成し、構成時に作成されたすべての DSP/BIOS オブジェクトのための適切な宣言を含むファイルに、そのヘッダを追加します（<program>.cfg.h）。アプリケーションのソース・ファイルにこのファイルをインクルードすることによって、DSP/BIOS オブジェクト宣言は完了します。

C54x プラットフォーム用の DSP/BIOS は、本来は、初期の C54x デバイスの 16 ビット・アドレッシング・モデル用に開発されたものです。新しい C54x デバイスには、far 拡張アドレッシング・モードが組み込まれており、DSP/BIOS にはこの環境で機能するように修正が加えられています。詳細については、アプリケーション・レポート『DSP/BIOS and TMS320C54x Extended Addressing』（文献番号 SPRA599）を参照してください。

1.3.5 メモリ・セグメント名

DSP/BIOS が使用するメモリ・セグメント名を 表 1-3 に示します。構成時のデフォルトのメモリ・セグメントの起点、サイズ、および名前を変更することができます。

表 1-3. メモリ・セグメント名

a. C54x プラットフォーム

セグメント	説明
IDATA	内部（オンデバイス）データ・メモリ
EDATA	外部データ・メモリの 1 次ブロック
EDATA1	外部データ・メモリの 2 次ブロック（EDATA と不連続）
IPROG	内部（オンデバイス）プログラム・メモリ
EPROG	外部プログラム・メモリの 1 次ブロック
EPROG1	外部プログラム・メモリの 2 次ブロック（EPROG と不連続）
USERREGS	ページ 0 ユーザ・メモリ（28 ワード）
BIOSREGS	ページ 0 予約レジスタ（4 ワード）
VECT	割り込みベクタ・セグメント

表 1-3.　メモリ・セグメント名（続き）

b. C55x プラットフォーム

セグメント	説明
IDATA	データ・メモリの 1 次ブロック
DATA1	データ・メモリの 2 次ブロック（DATA と不連続）
PROG	プログラム・メモリ
VECT	DSP 割り込みベクタ・テーブル・メモリ・セグメント

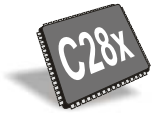
c. メモリ・セグメント名、C6000 EVM プラットフォーム

セグメント	説明
IPRAM	内部（オンデバイス）プログラム・メモリ
IDRAM	内部（オンデバイス）データ・メモリ
SBSRAM	CE0 上の外部 SBSRAM
SDRAM0	CE2 上の外部 SDRAM
SDRAM1	CE3 上の外部 SDRAM

d. メモリ・セグメント名、C6000 DSK プラットフォーム

セグメント	説明
SDRAM	外部 SDRAM

e. メモリ・セグメント名、C2800 DSK プラットフォーム



セグメント	説明
BOOTROM	ブート・コード・メモリ
FLASH	内部フラッシュ・プログラム・メモリ
VECT	VMAP=0 のときの割り込みベクタ・テーブル
VECT1	VMAP=1 のときの割り込みベクタ・テーブル
OTP	フラッシュ・レジスタ経由でワンタイム・プログラム可能メモリ
H0SARAM	内部プログラム RAM
L0SARAM	内部データ RAM
M1SARAM	内部ユーザ・タスク・スタック RAM

1.3.6 標準メモリ・セクション

構成は、表 1-4 に示すように標準メモリ・セクションとそのデフォルトの割り当てを定義します。これらのデフォルトの割り当ては、MEM マネージャを使用して変更することができます。詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「MEM Module」を参照してください。

表 1-4. 標準メモリ・セグメント

a. C54x プラットフォーム

セクション	セグメント
システム・スタック・メモリ（.stack）	IDATA
アプリケーション引数メモリ（.args）	EDATA
アプリケーション定数メモリ（.const）	EDATA
BIOS プログラム・メモリ（.bios）	IPROG
BIOS データ・メモリ（.sysdata）	EDATA
BIOS ヒープ・メモリ	IDATA
BIOS スタートアップ・コード・メモリ（.sysinit）	EPROG

b. C55x プラットフォーム

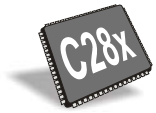
セクション	セグメント
システム・スタック・メモリ（.stack）、 システム・スタック・メモリ（.sysstack）	DATA
BIOS カーネル・ステート・メモリ（.sysdata）	DATA
BIOS オブジェクト、構成メモリ（.*obj）	DATA
BIOS プログラム・メモリ（.bios）	PROG
BIOS スタートアップ・コード・メモリ（.sysinit、.gblinit、.trcinit）	PROG
アプリケーション引数メモリ（.args）	DATA
アプリケーション・プログラム・メモリ（.text）	PROG
BIOS ヒープ・メモリ	DATA
2 次 BIOS ヒープ・メモリ	DATA1

表 1-4.　標準メモリ・セグメント（続き）

c. C6000 プラットフォーム

セクション	セグメント
システム・スタック・メモリ（.stack）	IDRAM
アプリケーション定数メモリ（.const）	IDRAM
プログラム・メモリ（.text）	IPRAM
データ・メモリ（.data）	IDRAM
スタートアップ・コード・メモリ（.sysinit）	IPRAM
C 初期化レコード・メモリ（.cinit）	IDRAM
初期化されない変数用のメモリ（.bss）	IDRAM

c. C2800 プラットフォーム



セクション	セグメント
システム・スタック・メモリ（.stack）	M1SARAM
プログラム・メモリ（.text）	IPROG
データ・メモリ（.data）	IDATA
アプリケーション定数メモリ（.const）	IDATA
スタートアップ・コード・メモリ（.sysinit）	IPROG
C 初期化レコード・メモリ（.cinit）	IDATA
初期化されない変数用のメモリ（.bss）	IDATA

1.4　詳細情報について

DSP/BIOS のコンポーネントおよび DSP/BIOS API のモジュールの詳細については、オンライン・ヘルプ・システムの「DSP/BIOS」、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』、『Code Composer Studio Online Tutorial』にある「DSP/BIOS の使用方法」のいずれかを参照してください。

第 2 章

プログラムの生成

本章では、DSP/BIOS を使用したプログラムの生成プロセスについて解説します。また、DSP/BIOS のコンポーネントにより生成されるファイル、またそれらのファイルの使用方法についても説明します。

項目 **ページ**

2.1 開発サイクル

DSP/BIOS は、反復的なプログラム開発サイクルをサポートします。ユーザは、アプリケーションの基本的なフレームワークを作成し、シミュレートされた処理負荷を使用して、DSP アルゴリズムを実際に適用する前にアプリケーションをテストすることができます。この方法で、各種の機能を実行するプログラム・スレッドの優先順位とタイプを簡単に変更することができます。

サンプルの DSP/BIOS 開発サイクルには次の手順が含まれていますが、特定のステップ、またはステップ・グループを反復して使用する場合もあります。

1) プログラムが使用する静的オブジェクトを構成します。この作業を行うには、Tconf スクリプト言語を使用します。

2) 構成スクリプトを実行します。これにより、プログラムをコンパイルしてリンクするときにインクルードするファイルが生成されます。

3) プログラムのフレームワークを作成します。C、C++、アセンブリ、またはこれらの言語を組み合わせて使用できます。

4) Code Composer Studio を使用して、プロジェクトにファイルを追加して、プログラムをコンパイルしてリンクします。

5) シミュレータまたは初期ハードウェア、および DSP/BIOS 解析ツールを使用して、プログラムの動作をテストします。ログとトレース、統計オブジェクト、タイミング、ソフトウェア割り込みなどを監視することができます。

6) プログラムが正しく実行されるようになるまで、ステップ 1 ～ 5 を繰り返します。この間に、基本プログラム構造に機能を追加したり変更を加えたりすることができます。

7) 実動ハードウェアの用意ができたら、実動ボードをサポートするように構成ファイルを変更し、そのボード上でプログラムをテストします。

2.2 DSP/BIOS アプリケーションを静的に構成する方法

1.2.2 項「DSP/BIOS 構成」（1-6 ページ）で説明したように、DSP/BIOS 構成を使用すると、実行時ではなく静的に、オブジェクトを作成したり、そのプロパティをセットしたりすることができます。構成を作成するには、GUI を介してかテキストを介してか、あるいはその 2 つの方法を組み合わせて使用します。

『DSP/BIOS Textual Configuration (Tconf) User's Guide』（文献番号 SPRU007）には、構成スクリプトで使用する構文に関する詳細が記述されています。

DSP/BIOS 構成を Code Composer Studio を構成するために使用する他の項目と混同しないようにしてください。これらの他の項目には、プロジェクト構成（通常、Debug または Release）、RTDX Configuration Control ウィンドウ、および CCS Setup ツール内でセットアップされるシステム構成が含まれています。

2.2.1 静的に作成された DSP/BIOS オブジェクトを参照する方法

プログラム内で参照される静的に作成されたオブジェクトは、すべての関数本体の外部で extern 変数として宣言する必要があります。たとえば、次のように宣言すると、PIP_Obj オブジェクトは、プログラム内でこのオブジェクトを定義した後に続くすべての関数で見えるようになります。

```
extern far  PIP_Obj inputObj;      /* C6000 devices  */
  または
extern      PIP_Obj inputObj;      /* C5000 and C2800 devices */
```

構成はこれらの定義を含むファイルを生成します。ファイルの名前は *cfg.h になります。ここで、* はプログラムの名前です。DSP/BIOS オブジェクトを参照する C ファイルでは #include の直後にこのファイルが記述されます。

2.2.1.1 C6000 のスモール・モデルとラージ・モデルの問題

DSP/BIOS 自体は、スモール・モデルを使用してコンパイルされていますが、DSP/BIOS アプリケーションをコンパイルするときに、C6000 コンパイラのスモール・モデルまたはラージ・モデルのバリエーションを使用することができます（『TMS320C6000 オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル』を参照）。実際には、アプリケーション・コード内で複数のコンパイル・モデルを混合使用することもできます。ただし、そのためには、B14 を基準とする相対変位を使用してアクセスされるすべてのグローバル・データが、.bss セクションの先頭から 32K バイト以内の位置に配置されている必要があります。

DSP/BIOS は、グローバル・データを .bss セクションに格納します。しかし、静的に構成されたオブジェクトは、.bss セクションには格納されません。したがって、アプリケーション・データを配置するときの柔軟性が非常に高くなります。たとえば、アクセス頻度の高い .bss はオンデバイス・メモリに入れ、大きくてアクセス頻度の低いオブジェクトは外部メモリに格納するといったことが可能です。

スモール・モデルでは、命令サイクル数を減らすために、いくつかの前提条件に基づいてグローバル・データが配置されます。グローバル・データのアクセスを最適化するためにスモール・モデル（デフォルトのコンパイル・モード）を使用している場合は、静的に作成されたオブジェクトが正しく参照されるように、コードに変更を加えることができます。

この問題に対処するには 4 つの方法があります。これらの手法についてはこの後で説明しますが、表 2-1 に示すように、どれにもそれぞれ長所と短所があります。

表 2-1. C6000 グローバル・オブジェクトを参照する方法

方法	far を使用したオブジェクト宣言	グローバル・オブジェクト・ポインタの使用	.bss に隣接するオブジェクト	ラージ・モデルを使用したコンパイル
コードがコンパイル・モデルに拘束されずに機能する能力	あり	あり	あり	あり
コードがオブジェクト配置に拘束されずに機能する能力	あり	あり	なし	あり
C コードを他のコンパイラに移植する能力	なし	あり	あり	あり
静的に作成されたオブジェクトのサイズが 32K バイトに拘束されない能力	あり	あり	なし	あり
.bss のサイズを最小限にする能力	あり	あり	なし	あり
命令サイクル数を最小限にする能力	なし（3 サイクル）	なし（2 ～ 6 サイクル）	あり（1 サイクル）	なし（3 サイクル）
オブジェクト当たりの記憶量を最小限にする能力	なし（12 バイト）	なし（12 バイト）	あり（4 バイト）	なし（12 バイト）
プログラムを記述しやすくデバッグが簡単	少々	エラーが発生しやすい	少々	あり

2.2.1.2 スモール・モデルで静的な DSP/BIOS オブジェクトを参照する方法

スモール・モデルでは、コンパイル済みのコードはすべて、データ・ページ・ポインタ・レジスタを基準とした相対位置によりグローバル・データにアクセスします。レジスタ B14 は、コンパイラにより読み出し専用レジスタとして扱われ、プログラムの起動時に .bss セクションの開始アドレスで初期化されます。グローバル・データは、.bss セクションの先頭から一定のオフセット位置にあるものと想定され、このセクションの長さは 32K バイト以下であるものと見なされます。したがって、次のような 1 つの命令により、グローバル・データにアクセスすることができます。

```
LDW  *+DP(_x), A0    ; load _x into A0 (DP = B14)
```

静的に作成されたオブジェクトは .bss セクションには格納されないので、スモール・モデルでコンパイルされたアプリケーション・コードがこれらのオブジェクトを正しく参照できるようにする必要があります。これには 3 つの方法があります。

- far キーワードを使用して、静的オブジェクトを宣言する。DSP/BIOS コンパイラは、C 言語に共通するこの拡張機能をサポートします。データ宣言内の far キーワードは、データが .bss セクションには含まれていないことを示します。

  たとえば、静的に作成された inputObj という名前の PIP オブジェクトを参照するには、このオブジェクトを次のように宣言します。

  ```
  extern far PIP_Obj inputObj;
  if (PIP_getReaderNumFrames(&inputObj)) {
       . . .
  }
  ```

- グローバル・オブジェクト・ポインタを作成し、初期化する。グローバル変数を作成し、初期化時にそのグローバル変数が、参照しようとしているオブジェクトのアドレスに設定されるようにすることができます。far キーワードを使用しなくても済むようにするには、このオブジェクトに対するすべての参照は、このポインタを使用して行う必要があります。次に例を示します。

  ```
  extern PIP_Obj inputObj;
  /* input MUST be a global variable */
  PIP_Obj *input = &inputObj;
  if (PIP_getReaderNumFrames(input)) {
       . . .
  }
  ```

  グローバル・ポインタを宣言し初期化するには、（オブジェクトの 32 ビット・アドレスを収容するため）余分なデータ・ワードが 1 つ必要になります。

  さらに、ポインタが静的変数または自動変数である場合は、この方法は無効です。次のコードは、スモール・モデルを使用してコンパイルした場合は期待したように機能しません。

  ```
  extern PIP_Obj inputObj;
  static PIP_Obj *input = &inputObj;   /* ERROR!!!! */
  if (PIP_getReaderNumFrames(input)) {
       . . .
  }
  ```

- すべてのオブジェクトを .bss に隣接する位置に配置する。すべてのオブジェクトが .bss セクションの終わりに配置されていて、それらのオブジェクトと .bss データを合わせた長さが 32K バイト未満であれば、.bss セクション内に割り当てられているものとしてそれらのオブジェクトを参照することができます。

  ```
  extern PIP_Obj inputObj;
  if (PIP_getReaderNumFrames(&inputObj)) {
       . . .
  }
  ```

  オブジェクトが確実にこの配置になるようにするには、構成スクリプトを使用して次のように設定します。

a) MEM オブジェクトを作成し、そのプロパティ（ベースと長さなど）をセットすることにより、新しいメモリ・セグメントを宣言します。あるいは、既存のデータ・メモリ MEM オブジェクトの 1 つを使用します。

b) スモール・モデル・コードにより参照されるすべてのオブジェクトを、このメモリ・セグメントに入れます。

c) 初期化されない変数メモリ（.bss）をこの同じセグメントに入れます。

2.2.1.3 ラージ・モデルで静的な DSP/BIOS オブジェクトを参照する方法

ラージ・モデルでは、コンパイル済みのコードはすべて、データにアクセスするときにまず 32 ビット・アドレス全体をアドレス・レジスタにロードし、次に LDW 命令の間接アドレッシング機能を利用してデータをロードします。次に例を示します。

```
MVKL     _x, A0     ; move low 16-bits of _x's address into A0
MVKH    _x, A0     ; move high 16-bits of _x's address into A0
LDW     *A0, A0    ; load _x into A0
```

ラージ・モデルのどのバリエーションでコンパイルしたアプリケーション・コードの場合でも、静的オブジェクトの位置により影響を受けることはありません。静的に作成されたオブジェクトを直接参照するすべてのコードがいずれかのラージ・モデル・オプションによってコンパイルされている場合は、コードはオブジェクトを通常のデータとして参照することができます。

```
extern PIP_Obj inputObj;
if (PIP_getReaderNumFrames(&inputObj)) {
     . . .
}
```

-ml0 ラージ・モデル・オプションは、すべての集合データが far と見なされるという点を除き、スモール・モデルと同じです。このオプションを使用すると、すべての静的オブジェクトは far オブジェクトと見なされますが、スカラ型（int、char、long など）は、near データとしてアクセスすることができます。したがって、多くのアプリケーションの場合、パフォーマンスの低下はほんのわずかで済みます。

2.3 DSP/BIOS オブジェクトを動的に作成する方法

代表的な DSP アプリケーションの場合、ほとんどのオブジェクトはプログラム実行全体で使用されるものなので、静的に作成する必要があります。デフォルト・オブジェクトの多くは、構成テンプレート内で自動的に定義されます。オブジェクトを静的に作成する利点は、次のとおりです。

- **DSP/BIOS 解析ツールでのアクセス効率が向上します。**「Execution Graph」には、静的に作成されたオブジェクトの名前が表示されます。さらに、静的に作成されたオブジェクトのみに関する統計情報も表示することができます。
- **コード・サイズを削減します。**代表的なモジュールの場合、XXX_create() 関数と XXX_delete() 関数には、そのモジュールを実装するために必要なコードの 50% が含まれています。TSK_create() および TSK_delete() への呼び出しを使用しないようにすれば、これらの関数の基礎コードはアプリケーション・プログラムにインクルードされません。他のモジュールについても同様です。オブジェクトを静的に作成すると、アプリケーション・プログラムのサイズを大幅に削減することができます。
- **実行時パフォーマンスが向上します。**オブジェクトの動的作成を回避することにより、コード・スペースを節約できるだけでなく、プログラムでシステムのセットアップを行うために消費する時間を短縮することができます。

オブジェクトを静的に作成する場合の制約は、次のとおりです。

- 静的オブジェクトは、必要であるかどうかに関係なく作成されます。発生頻度の低い実行時イベントが生じた場合にのみ使用されるようなオブジェクトは、動的に作成することができます。
- 静的オブジェクトは、XXX_delete() 関数を使用して実行時に削除することはできません。

多くの DSP/BIOS オブジェクト（すべてではありません）は、XXX_create() を呼び出すことにより作成できます（ここで XXX は特定モジュールの名前です）。静的にしか作成できないオブジェクトも一部あります。XXX_create() 関数は、オブジェクトの内部状態情報を格納するためのメモリを割り当て、XXX モジュールが提供する他の関数を呼び出すときに新規に作成されたオブジェクトを参照するために使用するハンドルを返します。

ほとんどの XXX_create() 関数では、最後のパラメータとして、新規に作成されたオブジェクトに属性を割り当てるために使用される XXX_Attrs 型の構造体を指すポインタを指定することができます。規約では、このパラメータがヌルの場合は、オブジェクトには一組のデフォルト値が割り当てられます。これらのデフォルト値は、ヘッダ・ファイル内にリストされている定数構造体 XXX_ATTRS に含まれています。ユーザは、まず XXX_Attrs 型の変数を初期化しておき、XXX_create() を呼び出す前にアプリケーション独自の属性に従ってその変数のフィールドを選択的に更新することができます。TSK_create() を使用して動的オブジェクトを作成するサンプル・コードを 例 2-1 に示します。

例 2-1. 動的オブジェクトの作成方法と参照方法

```
#include <tsk.h>
TSK_Attrs   attrs;
TSK_Handle  task;

attrs = TSK_ATTRS;
attrs.name = "reader";
attrs.priority = TSK_MINPRI;

task = TSK_create((Fxn)foo, &attrs);
```

XXX_create() 関数は、タスクのオブジェクトへのアドレスをハンドルとして渡します。このハンドルは、例 2-2 に示すように、たとえばオブジェクトを削除するなど、オブジェクトを参照するときに引数として渡すことができます。XXX_create() 関数により作成されたオブジェクトは、XXX_delete() 関数を呼び出して削除します。これにより、そのオブジェクトの内部メモリが解放され、後で使用できるようにシステムに戻されます。

グローバル定数 XXX_ATTRS を使用してデフォルト値をコピーし、この定数の各フィールドを更新し、引数として XXX_create() 関数に渡します。

例 2-2. 動的オブジェクトの削除方法

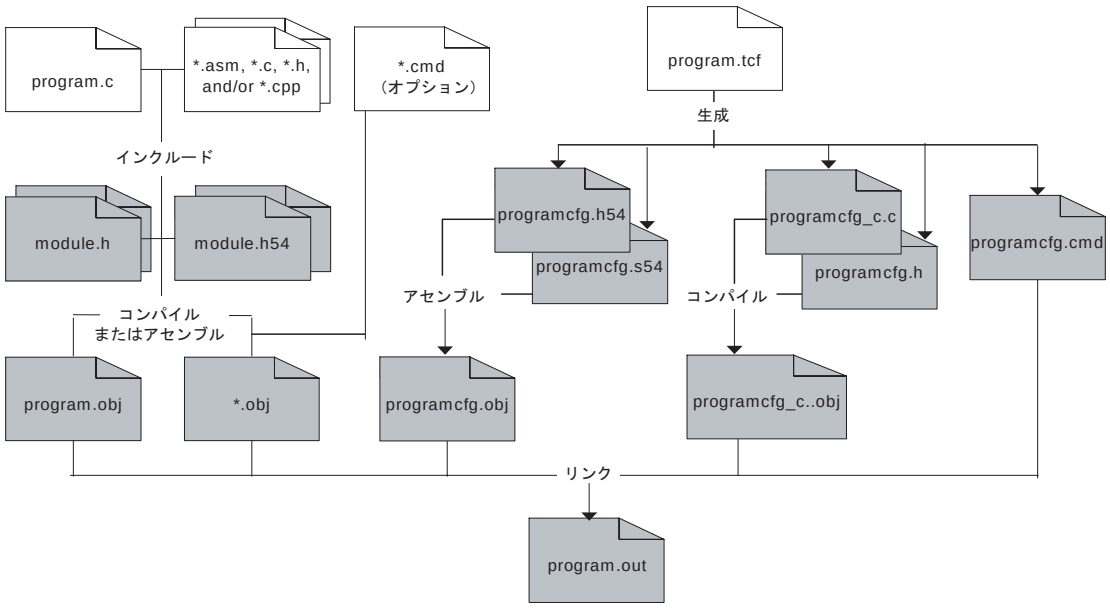
```
TSK_delete (task);
```

動的に作成された DSP/BIOS オブジェクトは、プログラム実行時に調整できます。

2.4 DSP/BIOS プログラムの作成に使用されるファイル

図 2-1 に、DSP/BIOS アプリケーションを作成するために使用するファイルを示します。背景が白で示されているのはユーザが作成するファイルであり、背景がグレーで示されているのは生成されるファイルです。*program* という語は、使用するプロジェクトまたはプログラムの名前を表します。54 という数字は、ご使用のプラットフォームに合わせて、それぞれ 28、55、64 と読み替えてください。

図 2-1. DSP/BIOS アプリケーションのファイル構成



プログラム・ファイル

- **program.c**：main 関数を含むプログラム・ソース・ファイル。追加の .c ソース・ファイルおよびプログラムに使用する .h ファイルも含めることができます。ユーザ関数については、2.9 節「DSP/BIOS により呼び出されるユーザ関数」を参照してください。
- **program.tcf :** 実行時に構成ファイルを生成する Tconf スクリプト。このファイルが構成のソースになります。このファイルを Code Composer Studio プロジェクトに追加して、アプリケーションの構成部分を作成します。
- ***.asm**：オプションのアセンブリ・ソース・ファイル。これらのファイルの 1 つには、C または C++ の main 関数の代わりに _main というアセンブリ言語関数を含めることができます。
- **module.h**：C または C++ プログラム用の DSP/BIOS API ヘッダ・ファイル。ソース・ファイルには、C プログラムで使用するすべてのモジュール用の std.h ファイルおよびヘッダ・ファイルをインクルードする必要があります。

- **module.h54** : アセンブリ・プログラムの DSP/BIOS API ヘッダ・ファイル。アセンブリ・ソース・ファイルには、アセンブリ・ソースで使用するすべてのモジュール用の *.h54 ヘッダ・ファイルをインクルードする必要があります。
- **program.obj :** ソース・ファイルを元にしてコンパイルまたはアセンブルされたオブジェクト・ファイル。
- ***.obj :** オプションのアセンブリ・ソース・ファイル用のオブジェクト・ファイル。
- ***.cmd** : DSP/BIOS 構成時に定義していない、ユーザ・プログラム用の追加セクションを含むオプションのリンカ・コマンド・ファイル。
- **program.out :** ターゲット用の実行可能プログラム（完全にコンパイル、アセンブル、リンクされたもの）。このプログラムは、Code Composer Studio のコマンドを使用してロードおよび実行することができます。

静的な構成ファイル

prog.gen() メソッドを含む Tconf スクリプトを実行すると、次のファイルが作成されます（ここで「program」は構成ファイルの名前で、54 という数字は、ご使用のプラットフォームに合わせて、それぞれ 28、55、64 と読み替えてください）。

- **programcfg.cmd :** DSP/BIOS のリンカ・コマンド・ファイル。このファイルは、DSP/BIOS 固有のリンク・オプションとオブジェクト名、および DSP プログラムの一般的なデータ・セクション（.text、.bss、.data など）を定義します。*.tcf ファイルを Code Composer Studio プロジェクトに追加すると、このファイルは Project View の Generated Files フォルダに自動的に追加されます。
- **programcfg.h** : DSP/BIOS モジュールのヘッダ・ファイルが含まれ、構成時に作成されたオブジェクトの外部変数を宣言します。
- **programcfg_c.c** : DSP/BIOS 関連のオブジェクトを定義します （CSL オブジェクトは定義しません）。
- **programcfg.s54 :** DSP/BIOS 設定用のアセンブリ言語ソース・ファイル。*.tcf ファイルを Code Composer Studio プロジェクトに追加すると、このファイルは Project View の Generated Files フォルダに自動的に追加されます。
- **programcfg.h54 :** programcfg.s54 にインクルードされるアセンブリ言語ヘッダ・ファイル。
- **program.cdb :** 実行時解析ツールが使用する構成設定ファイルを格納しています。以前のバージョンでは、これは構成ソース・ファイルでした。このファイルは、*.tcf ファイルを実行すると作成されます。このファイルは、DSP/BIOS 解析ツールが使用します。
- **programcfg.obj :** 構成スクリプトによって作成されるソース・ファイルを元に作成されるオブジェクト・ファイル。

2.5 プログラムのコンパイル方法とリンク方法

DSP/BIOS 実行プログラムをビルドするには、Code Composer Studio プロジェクト、または独自のメイクファイルを使用します。Code Composer Studio ソフトウェアには、GNU のメイク・ユーティリティ gmake.exe、およびチュートリアルの例をビルドするための gmake 用のサンプル・メイクファイルが含まれています。

ご使用の DSP/BIOS バージョン固有の詳細については、DSP/BIOS インストール時の GettingStartedGuide.html ファイルを参照してください。

2.6 DSP/BIOS でのランタイム・サポート・ライブラリの使用方法

構成によって生成されるリンカ・コマンド・ファイルには、DSP/BIOS、RTDX、およびランタイム・サポート・ライブラリなど必要なライブラリを検索するための疑似命令が、自動的にインクルードされます。ランタイム・サポート・ライブラリは、rts.src を元に作成されます。rts.src には、ランタイム・サポート関数のソース・コードが含まれています。これらの関数は、C 言語の一部にはなっていない標準 ANSI 関数です（メモリ割り当て、文字列変換、および文字列検索のための関数など）。rts.src の中で定義されているメモリ管理関数の多くは、DSP/BIOS ライブラリの中でも定義されています。このような関数には、malloc、free、memalign、calloc、および realloc があります。これらのライブラリは、それぞれ異なる実装をサポートしています。たとえば、DSP/BIOS のバージョンは MEM モジュールを含む形で実装されるので、DSP/BIOS API コール MEM_alloc および MEM_free を使用します。DSP/BIOS ライブラリは、ランタイム・サポート・ライブラリに含まれている機能と同じものを部分的に提供するので、DSP/BIOS リンカ・コマンド・ファイルには、表 2-2 に示すファイルを含まない rtsbios という名前の特殊バージョンのランタイム・サポート・ライブラリが含まれています。

表 2-2. rtsbios には含まれていないファイル

C54x プラットフォーム	C55x プラットフォーム	C6000 プラットフォーム
		
memory.c	memory.c	memory.c
autoinit.c	boot.c	sysmem.c
boot.c		autoinit.c
		boot.c

多くの DSP/BIOS プロジェクトでは、ライブラリの再読み取りを強制実行するために、–x リンカ・スイッチを使用する必要があります。たとえば、printf が malloc を参照していて、DSP/BIOS ライブラリから malloc がまだリンクされていない場合は、malloc への参照を解決するために、このリンカ・スイッチの働きにより DSP/BIOS ライブラリの再検索が強制実行されます。

ランタイム・サポート・ライブラリは、ブレークポイント付きの printf を実装しています。アプリケーションが printf を使用する頻度と呼び出しの頻度に応じて、printf() が RTDX に干渉し、その結果メッセージ・ログや統計ビューなどのリアルタイム解析ツールが影響を受け、これらのツールの更新が行われなくなる場合があります。これは、printf によるブレークポイント処理の方が RTDX より優先順位が高いからです。したがって、DSP/BIOS アプリケーション内では、可能な限り、printf に対する呼び出しの代わりに LOG_printf を使用することをお勧めします。

注：

DSP/BIOS アプリケーション内では、DSP/BIOS ライブラリ・バージョンの malloc、free、memalign、calloc、および realloc を使用することをお勧めします。アプリケーション内でこれらの関数を直接参照せずに、これらの関数の 1 つまたは複数を参照する別のランタイム・サポート関数を呼び出す場合は、「–u _symbol」（たとえば、–u _malloc）をリンカ・オプションに追加してください。–u リンカ・オプションは、シンボル（malloc など）を未解決のシンボルとしてリンカのシンボル・テーブルに導入します。その結果、リンカは、ランタイム・サポート・ライブラリからでなく DSP/BIOS ライブラリから、そのシンボルを解決します。何か疑問がある場合は、マップ・ファイルを調べて、アプリケーションのライブラリ・ソースに関する情報を入手することができます。

2.7 DSP/BIOS のスタートアップ・シーケンス

DSP/BIOS アプリケーション起動時のスタートアップ・シーケンスは、boot.s54 ファイル（C54x プラットフォームの場合）、または autoinit.c および boot.s*nn* ファイル（C6000 および C55x プラットフォームの場合）内の呼び出しまたは命令によって決まります。これらのファイルのコンパイル済みバージョンは、bios.a*nn* および biosi.a*nn* ライブラリに収めてあり、ソース・コードは製品付属の配布ディスクに入っています。ブート・ファイルのソース・コード内で指定されている DSP/BIOS スタートアップ・シーケンスを次に示します。このスタートアップ・シーケンスは変更しないでください。

1) **DSP を初期化します。**DSP/BIOS プログラムは、C または C++ 環境のエントリ・ポイント c_int00 から起動されます。リセット割り込みベクタは、リセット後に c_int00 に分岐するように設定されます。

C54x プラットフォームの場合は、c_int00 の始めでは、システム・スタック・ポインタ（SP）は .stack の終わりを指すように設定されます。st0 や st1 などのステータス・レジスタも初期化されます。

C55x プラットフォームの場合は、c_int00 の始めでは、データ（ユーザ）スタック・ポインタ（XSP）、およびシステム・スタック・ポインタ（XSSP）は、それぞれユーザ・スタックとシステム・スタックの終わりを指すように設定されます。さらに、XSP は偶数アドレス境界にアライメントされます。

C6000 プラットフォームの場合は、c_int00 の始めでは、システム・スタック・ポインタ（B15）およびグローバル・ページ・ポインタ（B14）は、それぞれスタック・セクションの終わりと .bss の始めを指すように設定されます。AMR、IER、および CSR などの制御レジスタも初期化されます。

2) **.cinit レコードから .bss を初期化します。**スタックが設定されると、初期化ルーチンが呼び出されて、.cinit レコードから取り出されたデータで変数を初期化します。

3) **BIOS_init を呼び出して、アプリケーションが使用するモジュールを初期化します。**BIOS_init は、基本モジュールの初期化を行います。BIOS_init は、アプリケーションが使用する各 DSP/BIOS モジュールごとに MOD_init マクロを起動します。BIOS_init は構成により生成され、programcfg.s*nn* ファイルに格納されます。

   - HWI_init は、ISTP および割り込みセレクタ・レジスタを設定し、C6000 プラットフォーム上の IER 内の NMIE ビットをセットし、すべてのプラットフォーム上の IFR をクリアします。詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「HWI Module」を参照してください。

注：

割り込み構成時に、DSP/BIOS は、対応する ISR（割り込みサービス・ルーチン）を、割り込みサービス・テーブル内の適切な位置にプラグインします。ただし、DSP/BIOS は IER の割り込みビットをイネーブルにはしません。これは、スタートアップの時点、およびアプリケーション実行中の必要な時点で、ユーザが行う必要があります。

- **HST_init は、ホスト入出力（I/O）チャネル・インターフェイスを初期化します。**このルーチンの詳細な仕様は、ホストからターゲットへのリンクに使用されている実装によって異なります。たとえば、C6000 プラットフォームでは、RTDX が使用されている場合、HST_init は、RTDX 用として予約されているハードウェア割り込みに該当する IER の ビットをイネーブルにします。

- **IDL_init は、アイドル・ループ命令カウントを計算します。**
  Idle Function Manager の構成で「Auto calculate idle loop instruction count」プロパティが真にセットされている場合は、IDL_init は、スタートアップ・シーケンス内のこの時点で、アイドル・ループ命令カウントを計算します。アイドル・ループ命令カウントは、CPU Load Graph に表示される CPU の負荷を調整するために使用されます (3.4.2 項「CPU の負荷」(3-20 ページ）を参照)。

4) **.pinit テーブルを処理します。**.pinit テーブルは、初期化関数を指すポインタから構成されています。C++ プログラムの場合は、.pinit 処理中にグローバル・オブジェクトのクラス・コンストラクタが実行されます。

5) **ユーザ・プログラムのメイン・ルーチンを呼び出します。**DSP/BIOS のすべてのモジュールの初期化手順が完了すると、ユーザ・プログラムのメイン・ルーチンが呼び出されます。このルーチンは、アセンブリ、C、C++、またはこれらの言語を組み合わせて記述することができます。C コンパイラは関数名の先頭に下線を付加するので、C または C++ では関数名として main を使用し、アセンブリでは関数名として _main を使用します。

   この時点では、まだハードウェア割り込みもソフトウェア割り込みもイネーブルになっていないので、ユーザのアプリケーション用の初期化手順（ユーザ独自のハードウェア初期化ルーチンの呼び出しなど）を、メイン・ルーチンから行うことができます。main 関数は個々の割り込みマスク・ビットをイネーブルにしますが、HWI_enable を呼び出して割り込みをグローバルにイネーブルにする必要はありません。

6) **BIOS_start を呼び出して DSP/BIOS を起動します。**BIOS_init と同様に、BIOS_start も構成により生成され、programcfg.*snn* ファイルに格納されます。BIOS_start は、メイン・ルーチンから復帰後に呼び出されます。BIOS_start が担当する作業は、DSP/BIOS モジュールをイネーブルにすること、および各 DSP/BIOS モジュールの MOD_startup マクロを起動することです。構成時に TSK Manager がイネーブルにされている場合は、BIOS_start への呼び出しは復帰しません。次に例を示します。

   - CLK_startup は、PRD レジスタを設定し、CLKCLK マネージャで選択されたタイマに対応する IER（C6000 プラットフォーム）または IMR（C5400 プラットフォーム）のビットをイネーブルにし、最後にそのタイマを起動します

（このマクロが展開されるのは、構成時に CLK マネージャをイネーブルにしてある場合だけです）。

- PIP_startup は、作成した各パイプ・オブジェクトごとに、notifyWriter 関数を呼び出します。
- SWI_startup は、ソフトウェア割り込みをイネーブルにします。
- HWI_startup は、C6000 プラットフォームでは CSR の GIE ビットをセットし、C5400 プラットフォームでは ST1 レジスタの INTM ビットをクリアすることにより、ハードウェア割り込みをイネーブルにします。
- TSK_startup は、タスク・スケジューラをイネーブルにし、実行可能な状態にある最も優先順位の高いタスクを起動します。アプリケーションに現在実行可能な状態にあるタスクがない場合は、TSK_idle が実行され、IDL_loop が呼び出されます。TSK_startup が呼び出されるとアプリケーションが開始されるので、実行は TSK_startup または BIOS_start から復帰しません。TSK_startup が実行されるのは、構成時にタスク・マネージャがイネーブルにされている場合だけです。

7) **アイドル・ループを実行します。**アイドル・ループに入るには、2 つの方法があります。第 1 の方法では、タスク・マネージャがイネーブルにされます。タスク・スケジューラが TSK_idle を実行し、その結果 IDL_loop が呼び出されます。第 2 の方法では、タスク・マネージャがディセーブルにされ、従って BIOS_start への呼び出しが復帰し、その後に IDL_loop への呼び出しが続きます。ブート・ルーチンは、IDL_loop を呼び出して、永続的に DSP/BIOS アイドル・ループに入ります。この時点で、ハードウェア割り込みとソフトウェア割り込みが可能になり、割り込みはアイドル実行より優先して実行されます。アイドル・ループはホストとの通信を管理するので、ホストとターゲット間のデータ転送が可能になります。

2.7.1 拡張スタートアップ：C5500 プラットフォームのみ

C5500 プラットフォームのアーキテクチャでは、ソフトウェアでレジスタ IVPD および IVPH をセットすることにより、ベクタ・テーブル（全体の長さは 256 バイト）の開始位置を再設定することができます。デフォルトでは、ハードウェア・リセットにより、これらのレジスタの両方に 0xFFFF がロードされ、リセット・ベクタは 0xFF00 ～ FF00 の位置からフェッチされます。ベクタ・テーブルを別の位置に移すには、ハードウェア・リセット**後に** IVPD と IVPH に目的のアドレスを書き込み、その上でソフトウェア・リセットを行う必要があります。この時点で、IVPD および IVPH の新しい値が有効になります。

マクロ HWI_init は、構成済みのベクタ・テーブル・アドレスを IVPD および IVPH にロードしますが、新しい IVPD および IVPH を有効にするには、その後でソフトウェア・リセットが必要です。

また、C5500 プラットフォームでは、他に 3 つのスタック・モードも使用できます（表 2-3 を参照）。デフォルト以外のいずれかのモードでプロセッサを構成するには、ユーザは Code Composer Studio のデバッガ・ツールを使用して、ビット 28 および 29 をリセット・ベクタの位置に適切にセットしてから、ソフトウェア・リセットを適用する必要があります。詳細については、『TMS320C55x DSP CPU Reference Guide』を参照してください。

表 2-3. C5500 プラットフォームのスタック・モード

スタック・モード	説明	リセット・ベクタの設定
2x16 高速リターン	SP/SSP 独立、高速リターン機能を実現するために RETA/CFCT を使用	XX00 : XXXX : <24 ビット・ベクタ・アドレス >
2x16 低速リターン	SP/SSP 独立、RETA/CFCT は不使用	XX01 : XXXX : <24 ビット・ベクタ・アドレス >
1x32 低速リターン（リセットのデフォルト）	SP/SSP 同期、RETA/CFCT は不使用	XX02 : XXXX : <24 ビット・ベクタ・アドレス >

さらに、DSP/BIOS 構成はアプリケーションが使用するモードを照合するために、HWI マネージャのスタック・モードのプロパティをセットする必要があります。詳細については、『TMS320C5000 DSP/BIOS API Reference Guide』を参照してください。

2.8 DSP/BIOS での C++ の使用方法

DSP/BIOS アプリケーションは C++ で記述することができます。C++ アプリケーションの開発を円滑に進めるには、C++ と DSP/BIOS の関連性について理解しておくことが重要です。必要な考慮事項としては、メモリ管理、ネーム・マングリング、構成プロパティからのクラス・メソッドの呼び出し、クラス・コンストラクタとクラス・デストラクタに関する特殊な考慮事項があります。

2.8.1 メモリ管理

関数 new および delete は、動的なメモリ割り当ておよび割り当て解除のための C++ 演算子です。DSP/BIOS アプリケーションでは、これらの演算子は、DSP/BIOS のメモリ管理関数 MEM_alloc および MEM_free を使用して実装されているので、再入可能です。ただし、メモリ管理関数は、呼び出し元のスレッドがメモリに対するロックを獲得していないと先へ進めないので、要求されたロックをすでに他のスレッドが保持している場合はブロッキングが発生します。したがって、new および delete は TSK オブジェクトだけが使用するようにしてください。

関数 new および delete は、DSP/BIOS ライブラリではなく、ランタイム・サポート・ライブラリにより定義されています。最初に検索されるのは DSP/BIOS ライブラリなので、アプリケーションによっては、rtsbios（ランタイム・サポート）ライブラリ内で最初に参照されているシンボルの中に未定義のものがあることを示す、リンカ・エラーが生じることがあります。このリンカ・エラーを回避するには、–x リンカ・オプションを使用して、未定義の参照を解決するためにライブラリの再検索が行われるようにします（詳細については、2.6 節「DSP/BIOS でのランタイム・サポート・ライブラリの使用方法」を参照）。

2.8.2 ネーム・マングリング

C++ コンパイラは、関数のリンクレベル名の中でその関数のシグニチャを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、および型の安全なリンクを実現します。リンク名にシグニチャをエンコードするプロセスをネーム・マングリングといいます。ネーム・マングリングは、DSP/BIOS アプリケーションと干渉し合うことがあります。これは、構成内では、関数名を使用して C++ ソース・ファイル内で宣言されている関数を参照するからです。ネーム・マングリングを防止して、構成内で関数を認識できるようにするには、例 2-3 の部分コードに示すように、extern C ブロック内で関数を宣言する必要があります。

例 2-3. extern C ブロック内での関数の宣言方法

```
extern "C" {
Void function1();
Int function2();
}
```

これにより、構成内で関数を参照することができます。たとえば、SWI 通知のたびに function1() を実行する必要のある SWI オブジェクトがあるとすれば、その SWI オブジェクトの関数プロパティに function1 を入力します。

extern C ブロック内で宣言された関数は、ネーム・マングリングの影響を受けません。関数オーバーロードはネーム・マングリングにより達成されるので、構成スクリプトから呼び出される関数の場合は、関数オーバーロードに制約があります。つまり、extern C ブロックに含めることができるのは、オーバーロードされた関数の 1 つのバージョンだけです。したがって、例 2-4 に示すコードはエラーになります。

例 2-4. 関数オーバーロードの制約

```
extern “C” {
Int addNums(Int x, Int y);
Int addNums(Int x, Int y, Int z); // error, only one version
                                  // of addNums is allowed
}
```

DSP/BIOS C++ アプリケーションでは名前オーバーロードを使用できますが、構成スクリプトから呼び出すことができるのはオーバーロードされた関数の 1 つのバージョンだけです。

構成スクリプトから呼び出される関数では使用できない C++ 機能の 1 つに、デフォルト・パラメータがあります。C++ では、関数宣言の中で仮パラメータにデフォルト値を指定することができます。しかし、構成スクリプトから呼び出される関数はパラメータ値を指定する必要があります。値を指定しなかった場合、実パラメータ値は未定義になります。

2.8.3 構成からクラス・メソッドを呼び出す方法

多くの場合、構成内で参照する関数は、クラス・オブジェクトのメンバ関数となっています。このようなメンバ関数を構成から直接呼び出すことはできませんが、wrapper 関数を使用してこれと同じアクションを達成することができます。クラス・インスタンスをパラメータとして受け入れる wrapper 関数を記述することにより、wrapper の中からクラス・メンバ関数を起動することができます。

クラス・メソッドの wrapper 関数を例 2-5 に示します。

例 2-5.　クラス・メソッドの wrapper 関数

```
Void wrapper (SampleClass myObject)
{
   myObject->method();
}
```

クラス・メソッドが必要とするその他のパラメータも、wrapper 関数に渡すことができます。

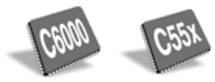
2.8.4　クラス・コンストラクタとデストラクタ

C++ クラス・オブジェクトがインスタンス化されるたびに、クラス・コンストラクタが実行されます。同様に、クラス・オブジェクトが削除されるたびに、クラス・デストラクタが呼び出されます。したがって、コンストラクタとデストラクタを記述するときは、これらの関数が実行される時点を予測し、それに従って関数を調整する必要があります。そのためには、クラス・コンストラクタおよびデストラクタが起動されるときに、どのようなタイプのスレッドが実行されるかを考慮することが重要です。

各種の DSP/BIOS スレッド（タスク、ソフトウェア割り込み、ハードウェア割り込み）から、それぞれどのような DSP/BIOS API 関数を呼び出すことができるかについては、さまざまなガイドラインが適用されます。たとえば、ソフトウェア割り込みのコンテキストの中からは、MEM_alloc や MEM_calloc などのようなメモリ割り当て API を呼び出すことはできません。したがって、あるクラスがソフトウェア割り込みによりインスタンス化される場合は、そのクラスのコンストラクタではメモリ割り当てを行わないようにする必要があります。同様に、クラス・デストラクタの実行が予測される時点も考慮することが重要です。クラス・デストラクタは、オブジェクトが明示的に削除されたときだけでなく、ローカル・オブジェクトが有効範囲から外れたときにも実行されます。クラス・デストラクタが呼び出されるときにどのタイプのスレッドが実行されているかを判別し、そのスレッドにとって適切な DSP/BIOS API コールのみを行うようにする必要があります。関数が呼び出し可能かどうかに関する詳細は、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』を参照してください。

2.9 DSP/BIOS により呼び出されるユーザ関数

DSP/BIOS オブジェクト（IDL、TSK、SWI、PIP、PRD、および CLK オブジェクト）により呼び出されるユーザ関数は、一定の規則に従っている必要があります。これは、レジスタが適切に使用され、複数の関数呼び出しにわたって値の有効性が維持されるようにするためです。



C6x および C55x プラットフォームでは、DSP/BIOS オブジェクトにより呼び出されるすべてのユーザ関数は、それぞれのプラットフォームにおける C コンパイラのレジスタ規則に従っていることが必要です。これは、C 言語とアセンブリ言語のどちらで記述された関数にも適用されます。

コンパイラは、すべての C 関数名には先頭に下線が付き、アセンブリ関数名には下線が付かないという規則に基づいて、C 関数とアセンブリ関数を区別します。C54x プラットフォームでは、C 関数とアセンブリ関数にはそれぞれ異なる規則が適用されるので、特にこの区別が重要な意味を持ちます。下線が前に付く関数（これには、C 関数と、下線で始まるアセンブリ関数が含まれます）は、C コンパイラの規則に従っている必要があります。C54x プラットフォームでは、アセンブリ関数（下線で始まらない関数）は、次の規則に従う必要があります。

- ❏ 最初の引数はレジスタ AR2 に入れて渡される。
- ❏ 2 番目の引数はレジスタ A に入れて渡される。
- ❏ 返り値はレジスタ A に入れて渡される。

注：

上記の規則は、TSK オブジェクトにより呼び出されるユーザ関数には適用されません。TSK オブジェクトにより呼び出されるすべてのユーザ関数（C とアセンブリの両方）には、C レジスタの規則が適用されます。

C54x プラットフォームでは、C 関数（または名前の先頭に下線が付くアセンブリ関数）を実行するときは、CPL（コンパイラ・モード）ビットがセットされていることが必要です。アセンブリ関数（名前が下線で始まっていないもの）の実行が開始されるときは、CPL ビットはクリアされ、復帰時もクリアされる必要があります。

C レジスタ規則の詳細については、ご使用のプラットフォームに対応した『オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル』を参照してください。

2.10 main から DSP/BIOS API を呼び出す方法

DSP/BIOS アプリケーション内の main ルーチンは、ペリフェラルを構成したり、ハードウェア割り込みを個別にイネーブルしたりする、ユーザによる初期化を目的とするものです。main は、DSP/BIOS のどのスレッド型（HWI、SWI、TSK、IDL）にも属さないので、プログラムの実行が main に達したときに、DSP/BIOS の初期化がすべて完了しているわけではないという点に注意する必要があります。これは、DSP/BIOS の初期化は 2 段階に分けて行われるからです。それは、main の前に実行される BIOS_init の時点と、プログラムが main から復帰した後で実行される BIOS_start の時点です。

DSP/BIOS API コールの中には、BIOS_start の初期化がまだ実行されていないため main ルーチンからは呼び出すことができないものがいくつかあります。BIOS_start が担当する作業は、グローバル割り込みをイネーブルにし、タイマを構成および起動し、スケジューラをイネーブルにして、DSP/BIOS スレッドを実行できるようにすることです。したがって、main からの実行に適さない DSP/BIOS 呼び出しとは、ハードウェア割り込みおよびタイマがイネーブルにされていることが前提条件となる API か、または実行をブロックするようなスケジューリング呼び出しを行う API です。たとえば、CLK_gethtime や CLK_getltime などの関数は、まだタイマが実行されていないので main から呼び出すのは不適切です。HWI_disable と HWI_enable も、まだハードウェア割り込みがグローバルにイネーブルにされていないので、呼び出さないようにしてください。SEM_pend または MBX_pend などのようにブロックを起こす可能性のある呼び出しも、スケジューラがまだ初期化されていないので main から呼び出さないでください。TSK_disable、TSK_enable、SWI_disable、SWI_enable などのスケジューリング呼び出しも、main 内で使用するのは不適切です。

BIOS_init は、main の前に実行され、MEM モジュールの初期化を担当します。したがって、動的メモリ割り当て関数を main から呼び出すのは問題ありません。MEM モジュール関数（MEM_alloc、MEM_free など）だけでなく、TSK_create や TSK_delete など、DSP/BIOS オブジェクトの動的作成および削除のための API も使用できます。

ブロックする呼び出しは main からは使用できませんが、DSP/BIOS スレッドを実行可能な状態にするためのスケジューリング呼び出しは使用できます。このような呼び出しには、SEM_post や SWI_post などがあります。このような呼び出しを main から行った場合、実行可能にされたスレッドは、プログラムが main から復帰し、BIOS_start の実行が完了した後で実行されるようにスケジュールされます。

特定の DSP/BIOS 関数呼び出しの詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』を参照してください。「Constraints and Calling Context」に、main から呼び出すことができる API かどうかが示されています。

第 3 章

計測

DSP/BIOS は、リアルタイムでプログラム解析を行うための明示的な方法と暗黙的な方法の両方を提供します。これらの方法は、アプリケーションのリアルタイム・パフォーマンスへの影響を最小限に抑えるように設計されています。

項目 **ページ**

3.1 リアルタイム解析

リアルタイム解析とは、システムのリアルタイム・オペレーション中に取得したデータを解析することです。その目的は、システムがその設計上の制約の範囲内で稼働しているかどうか、パフォーマンス目標を達成しているかどうか、さらに開発の余地が残されているかどうか判別しやすくすることです。

注 :

RTDX は、新しい DSP デバイスまたはボードの初期のリリースではサポートされていないことがあります。RTDX がサポートされていないプラットフォームでは、計測データは停止モードでのみ更新されます。すなわち、ターゲット・プログラムが実行中のときは、計測データはホスト PC には送られません。ターゲットを停止するか、ブレークポイントに到達すると、解析データが Code Composer Studio で表示できるように転送されます。

3.1.1 リアルタイム・デバッグと循環式デバッグ

順次ソフトウェアのデバッグでは、エラーが発生するまでプログラムを実行するというのが伝統的な手法です。そして、エラーが発生した時点で実行を停止し、プログラムの状態を調べ、ブレークポイントを挿入し、プログラムを再実行して情報を収集します。このような循環式デバッグは、非リアルタイムの順次ソフトウェアに対しては効果があります。しかし、リアルタイム・システムではほとんど効果がありません。これは、リアルタイム・システムには連続稼働、非決定性要素に基づく実行、および厳密な時間的制約という特徴があるからです。

DSP/BIOS 計測 API および DSP/BIOS 解析ツールは、循環式デバッグ・ツールの機能を補完する形で、稼働中のリアルタイム・システムを監視できるように設計されています。このリアルタイムの監視データを使用してリアルタイム・システムの稼働状況を調べ、システムのデバッグとパフォーマンス調整を効率的に行うことができます。

3.1.2 ソフトウェア計測とハードウェア計測

ソフトウェアによる監視は、ターゲット・アプリケーションの一部である計測コードから構成されています。このコードは実行時に実行され、重要なイベントに関するデータがターゲット・システムのメモリに保存されます。このように、計測コードではターゲット・システムの計算パワーとメモリの両方を利用します。

ソフトウェア計測には、柔軟性が高くハードウェアを追加する必要がないという利点があります。ただし、この計測はターゲット・アプリケーションの一部なので、パフォーマンスとプログラムの動作が影響を受けることがあります。ハードウェアによる監視を行わない場合には、プログラムに与える影響と十分な情報を収集するという目的との間のバランスをとるという問題に直面することになります。計測が制限されると十分な情報を入手することができず、過剰な計測は測定対象のシステムに過大な性能低下を生じさせる恐れがあります。

DSP/BIOS は、実行に対する影響と情報収集との間のバランスを精密に制御するためのさまざまなメカニズムを提供しています。しかも、DSP/BIOS 計測のすべてのオペレーションには一定の短い実行時間が必要です。オーバーヘッド時間が一定なので、計測の影響を事前に知ることができ、測定の要素から除外できます。

3.2 計測のパフォーマンス

暗黙の DSP/BIOS 計測がすべてイネーブルの場合、一般的なアプリケーションでは、1% 未満の範囲内で CPU の負荷が増大します。計測がアプリケーションのパフォーマンスに与える影響を最小限に抑えるために、次の手法が用いられています。

- ターゲットとホストの間の計測通信は、バックグラウンド（IDL）スレッドの中で行われます。このスレッドは優先順位が最も低いので、計測データの通信がアプリケーションのリアルタイム動作に影響を与えることはありません。
- ホストのユーザは、ホストがターゲットをポーリングするレートを制御することができます。ターゲットとの間の不要な外部相互作用をすべて排除したいときには、ホストとターゲット間の相互作用をすべて停止することができます。
- ターゲットは、トレースがイネーブルになっている場合以外は、「Execution Graph」（実行グラフ）、または暗黙的な統計情報を保存しません。TRC モジュールと、予約済みトレース・マスク（TRC_USER0 および TRC_USER1）の 1 つを使用して、アプリケーションの明示計測をイネーブルまたはディセーブルにすることもできます。
- ログおよび統計データは、常にホストで書式化されます。STS オブジェクトおよび CPU の負荷の平均値は、ホストで計算されます。「Execution Graph」を表示するために必要な計算もホストで行われます。

- ❑ 次に示すように、LOG、STS、および TRC モジュールの動作は非常に速く、しかも常に一定の時間で実行されます。

  - ■ LOG_printf および LOG_event：約 30 命令
  - ■ STS_add：約 30 命令
  - ■ STS_delta：約 40 命令
  - ■ TRC_enable および TRC_disable：約 4 命令

  - ■ LOG_printf および LOG_event：約 25 命令
  - ■ STS_add：約 10 命令
  - ■ STS_delta：約 15 命令
  - ■ TRC_enable および TRC_disable：約 4 命令

  - ■ LOG_printf および LOG_event：約 32 命令
  - ■ STS_add：約 18 命令
  - ■ STS_delta：約 21 命令
  - ■ TRC_enable および TRC_disable：約 6 命令

- ❑ 1 つの STS オブジェクトに使用されるデータ・メモリの量は、C5000 プラットフォームの場合 8 ワード、C6000 プラットフォームの場合 4 ワードにすぎません。これは、ホストは、統計オブジェクトからデータをアップロードするときに、常にこれと同数のワードを転送することを意味します。

- ❑ 統計情報は、ターゲットでは 32 ビット変数、ホストでは 64 ビット変数の中で累算されます。ホストは、ターゲットをポーリングしてリアルタイムの統計情報を取得すると、ターゲット上の変数をリセットします。したがって、ターゲットで必要な空間は最小限に抑えられ、しかも長期間のテスト実行に関する統計情報を維持することができます。

- ❑ LOG オブジェクトのバッファ・サイズを指定することができます。バッファ・サイズは、プログラムのデータ・サイズ、およびログ・データのアップロードに必要な時間に影響を与えます。

- ❑ パフォーマンス上の理由により、デフォルトでは暗黙のハードウェア割り込みの監視はディセーブルにされています。これがディセーブルにされているときは、パフォーマンスへの影響はまったくありません。イネーブルにされているときは、統計オブジェクト内のデータが更新されると、監視対象の各割り込みについて 1 回の割り込みが発生するたびに、20 ～ 30 命令を実行することになります。

3.2.1 計測付きカーネルと計測なしカーネル

アプリケーションのグローバルなプロパティを変更することにより、カーネル計測に対するサポートをディセーブルにすることができます。GBL モジュールには、「Enable Real Time Analysis」というプロパティがあります。このプロパティを偽にセットすると、最適のコード・サイズと実行速度を達成できます。この状況は、暗黙計測をサポートしない DSP/BIOS ライブラリにリンクすることで実現されます。ただし、これに伴って、DSP/BIOS 解析ツールに対するサポートや、LOG、TRC、および STS モジュール API などの明示計測も抑止されます。

表 3-1 に、計測なしカーネルと計測付きカーネルを比べた場合の、コード・サイズの増加の例を示します。数値は、計測付きカーネルを使用した場合に予測されるコードの増加量についての一般的な目安を示すものです。表 3-1 に、Code Composer Studio ソフトウェアに付属している DSP/BIOS の多くの機能を使用する 2 つのプロジェクト例をサンプルとして使用しています。サンプル・アプリケーションでは、DSP/BIOS のモジュールを組み込むことにより計測コードを組み込んでいます。したがって、表に示す数値は、計測により発生するコード・サイズの増加量を示すものであり、ユーザのアプリケーション間のサイズの相違やバリエーションによって影響を受けるものではありません。第 1 例の Slice には TSK、SEM、および PRD モジュールが組み込まれており、第 2 例の Echo は PRD および SWI モジュールを使用しています。どちらのアプリケーション例も、特にコード・サイズを最小化するための設計は施されていません。

計測付きカーネルと計測なしカーネルの比較を含む、DSP/BIOS カーネルのパフォーマンス・ベンチマークについては、アプリケーション・レポート（文献番号 SPRA662）『DSP/BIOS Timing Benchmarks on the TMS320C6000 DSP』を参照してください。

表 3-1.　計測付きカーネルによるコード・サイズの増加の例

a. 例：Slice

説明（サイズはすべて MADU 数）	C54x プラットフォーム 	C55x プラットフォーム 	C6000 プラットフォーム 
計測なしカーネルの場合のサイズ	12,500	32,000	78,900
計測付きカーネルの場合のサイズ	14,350	33,800	86,600
計測付きカーネルの場合のサイズの増加	1,850	1,800	7,700

b. 例：Echo

説明（サイズはすべて MADU 数）	C54x プラットフォーム 	C55x プラットフォーム 	C6000 プラットフォーム 
計測なしカーネルの場合のサイズ	11,600	41,200	68,800
計測付きカーネルの場合のサイズ	13,000	42,800	76,200
計測付きカーネルの場合のサイズの増加	1,400	1,600	7,400

3.3 計測 API

効果的な計測には、データを収集するオペレーションと、プログラム・イベントに応じてデータの収集を制御するオペレーションが必要です。DSP/BIOS は、データ収集のために次の 3 つの API モジュールを提供しています。

- **LOG（Event Log Manager: イベント・ログ・マネージャ）:** ログ・オブジェクトは、イベントに関する情報をリアルタイムでキャプチャします。システム・イベントは、システム・ログにキャプチャされます。追加のログを作成できます。ユーザ・プログラムは、どのログに対してもメッセージを追加することができます。
- **STS（Statistics Object Manager: 統計オブジェクト・マネージャ）:** 統計オブジェクトは、変数に関するカウント、最大値、および合計値をリアルタイムでキャプチャします。SWI（ソフトウェア割り込み）、PRD（期間）、HWI（ハードウェア割り込み）、PIP（パイプ）、および TSK（タスク）オブジェクトに関する統計情報を、自動的にキャプチャすることができます。さらに、ユーザ・プログラムにより、その他の統計情報をキャプチャするための統計オブジェクトも作成することができます。
- **HST（Host Channel Manager：ホスト・チャネル・マネージャ）:** 第 7 章「入出力（I/O）の概要とパイプ」で説明するホスト・チャネル・オブジェクトを使用することにより、プログラムは、生データ・ストリームを解析するためにホストに送信できます。

LOG および STS は、高い頻度で発生するリアルタイムのイベント・シーケンスや急激に変化するデータ値の統計要約情報のサブセットをキャプチャするために効率的な手段を提供します。このようなイベントが発生したり値が変化したりする速度は非常に早いので、シーケンス全体をホストに転送することが不可能な場合（帯域幅の制約のため）や、このシーケンスをホストに転送するためのオーバーヘッドがプログラムのオペレーションを妨害する場合があります。したがって DSP/BIOS には、他のモジュールが提供するデータ収集メカニズムを制御するための TRC（トレース・マネージャ）モジュールもあります。TRC モジュールは、各種のイベントおよび統計情報を、リアルタイムでキャプチャするのか DSP/BIOS 解析ツール を使用して対話式にキャプチャするのか制御します。

データの収集を制御することは重要です。これによって、プログラムの動作に対する計測の影響を抑制し、LOG および STS オブジェクトに必要な情報が含まれるようにし、実行時にイベントとデータ値の記録を開始または停止できるからです。

3.3.1 明示計測と暗黙計測

計測 API のオペレーションは、アプリケーションによって明示的に呼び出されるように設計されています。LOG モジュールのオペレーションを使用すると、任意のログに明示的にメッセージを書き込むことができます。STS モジュールのオペレーションを使用すると、データ変数またはシステム・パフォーマンスに関する統計情報を保存することができます。TRC モジュールを使用すると、プログラム・イベントに応じてログおよび統計情報のトレースをイネーブルまたはディセーブルにすることができます。

LOG および STS API は、DSP/BIOS の内部でプログラムの実行に関する情報を収集するときにも使用することができます。DSP/BIOS ルーチンの中でこのような内部呼び出しにより、暗黙計測サポートが提供されます。したがって、DSP/BIOS の計測用 API に対する明示的な呼び出しを含んでいないアプリケーションでも、DSP/BIOS 解析ツール を使用して監視し、解析することができます。たとえば、ソフトウェア割り込みの実行は、LOG_system という LOG オブジェクトに記録されます。

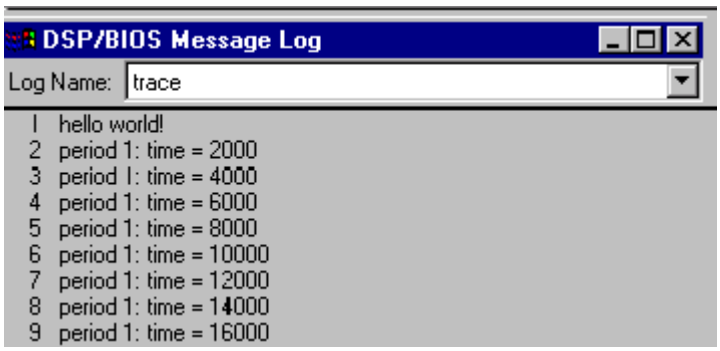
さらに、STS オブジェクトには、ソフトウェア割り込みに関するレディから完了までの最悪の場合の所要時間と、総合的な CPU の負荷の累計値が入ります。また、「Execution Graph」にはシステム・ティックの発生も表示されます。収集できる暗黙計測については、3.3.4.2 節「暗黙計測の制御」（3-16 ページ）を参照してください。

3.3.2 イベント・ログ・マネージャ（LOG モジュール）

このモジュールは、LOG オブジェクトを管理します。LOG オブジェクトには、ターゲット・プログラム実行中にリアルタイムでイベントがキャプチャされます。ユーザは、「Execution Graph」を使用したり、またはユーザ定義のログを表示することができます。

ユーザ定義のログには、ユーザ・プログラムが LOG_event オペレーションおよび LOG_printf オペレーションを使用して保存した情報がすべて含まれます。これらのログ内のメッセージは、図 3-1 に示すように、メッセージ・ログを使用してリアルタイムで表示することができます。メッセージ・ログにアクセスするには、「DSP/BIOS」→「Message Log」の順に選択します。

図 3-1. 「Message Log」ダイアログ・ボックス



「Execution Graph」（システム・ログ）は、個々のプログラム・コンポーネントの動作を示すグラフとして表示することもできます。

ログは、固定方式または循環方式のどちらにでもできます。この使い分けは、プログラムに基づいてロギングをイネーブルおよびディセーブルにするアプリケーションの場合に、重要な意味をもちます（3.3.4 項「トレース・マネージャ（TRC モジュール）」（3-15 ページ）で説明する TRC モジュール・オペレーションを使用）。

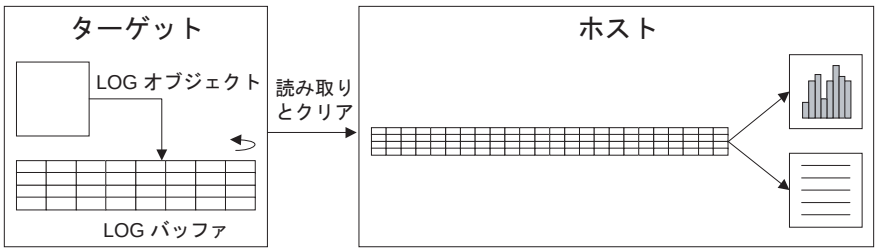
- **固定 :** ログは受け取った順にメッセージを保存し、メッセージ・バッファが満杯になるとメッセージの受け入れを停止します。したがって、固定ログには、ログがイネーブルにされて以降に発生した最初のイベントが保存されます。
- **循環 :** このログは、バッファが満杯になると古い方のメッセージを自動的に上書きします。したがって、循環ログには最後に発生したイベントが保存されます。

LOG オブジェクトを静的に構成して、メッセージ・バッファの長さや位置などのプロパティを割り当てます。

各メッセージ・バッファの長さは、ワード単位で指定します。1 つのメッセージについて、ログのバッファ内の 4 ワード分の記憶域が使用されます。先頭ワードにはシーケンス番号が入ります。メッセージ構造体の残りの 3 ワードには、イベントに依存したコード、および LOG_event（LOG オブジェクトに新しいイベントを追加する）などのオペレーションにパラメータとして渡されたデータ値が入ります。

図 3-2 に示すように、LOG バッファはターゲットから読み出されて、ホスト上のはるかに大きいバッファに保存されます。ホストにコピーされたレコードには、空のマークが付けられます。

図 3-2.　LOG バッファ・シーケンス

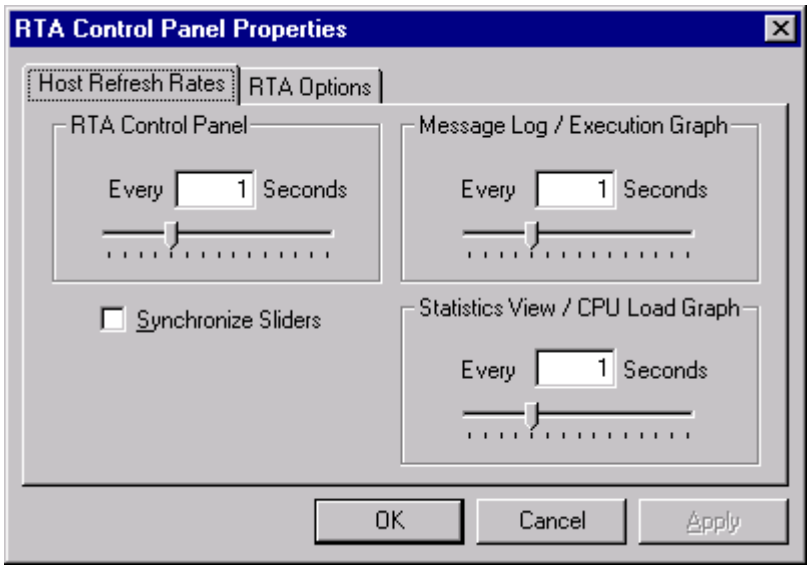


LOG_printf は、メッセージ構造体の 4 番目のワードを書式文字列（たとえば「%d, %d」）のオフセットまたはアドレスとして使用します。ホストは、この書式文字列と残りの 2 つのワードを使用して、データを表示用に書式化します。これにより、実際の printf オペレーション（およびこのオペレーションを行うコード）はホストで処理されるため、ターゲットで必要な時間とコード空間が最小限に抑えられます。

LOG_event と LOG_printf は、どちらも割り込みがディセーブルにされた状態でログに対するオペレーションを行います。したがって、優先順位の異なるハードウェア割り込みおよびその他のスレッドは、同期について配慮せずに同じログに書き込むことができます。

図 3-3 に示す「RTA Control Panel Properties」ダイアログ・ボックスを使用すると、ホストからターゲットをポーリングしてログ情報を要求する頻度を制御することができます。「RTA Control Panel」にアクセスするには、「DSP/BIOS」→「RTA Control Panel」の順に選択します。「RTA Control Panel」を右クリックし「Property Page」を選択して、リフレッシュ・レートをセットします。リフレッシュ・レートを 0 にセットした場合は、ユーザがログ・ウィンドウで右クリックしポップアップ・メニューから「Refresh Window」を選択しない限り、ホストがログ情報を要求するためにターゲットをポーリングすることはありません。また、ポップアップ・メニューを使用して、ログ情報を要求するポーリングを休止したり再開したりすることもできます。

図 3-3.　「RTA Control Panel Properties」ダイアログ・ボックス



メッセージ・ログ・ウィンドウに表示されるログ・メッセージには、イベントが発生した順序を示す番号が（トレース・ウィンドウの左側の列に）付けられます。これらの番号は、0 から始まる昇順のシーケンス番号です。ログが満杯になることがまったくない場合は、ログ・サイズを小さくすることができます。循環ログの場合は、十分な長さを設定しておくか頻繁にポーリングしないようにしておかないと、一部のログ・エントリがポーリングされる前に上書きされて失われてしまう恐れがあります。その場合は、ログ・メッセージ番号にギャップが生じます。ログ・メッセージに新たなシーケンス番号を追加してみると、ログ・エントリが失われているかどうかが明確になります。

LOG オブジェクトとそのパラメータに関する説明は、オンライン・ヘルプに含まれています。LOG モジュールの API コールについては、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「LOG Module」を参照してください。

3.3.3 統計オブジェクト・マネージャ（STS モジュール）

このモジュールは、プログラムの実行中に主要な統計情報を保存する統計オブジェクトを管理します。

個々の統計オブジェクトを静的に構成します。1 つの STS オブジェクトには、任意 32 ビット幅データ系列に関する次の統計情報が累積されます。

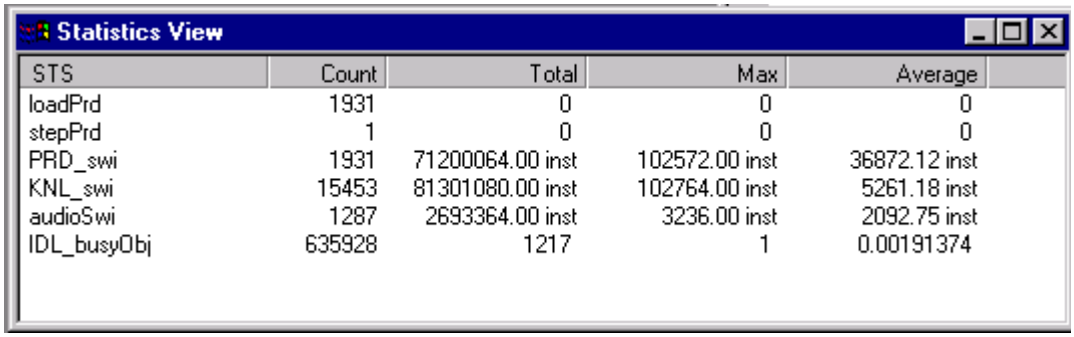
- **Count :** ターゲット上の、アプリケーション提供のデータ系列内の値の数。
- **Total** : ターゲット上の、この系列内の個々のデータ値の合計。
- **Maximum :** ターゲット上で、この系列内でこれまでに発生した最大値。
- **Average :**「Statistics View」解析ツールは、count と total を使用してホスト上の平均値を計算します。

STS_add オペレーションを呼び出すと、調査対象のデータ系列の統計オブジェクトが更新されます。たとえば、ソフトウェア割り込み解析アルゴリズムの中のピッチとゲイン、または閉ループ制御アルゴリズムの中の予測されるエラーと実際のエラーなどを調べることができます。

DSP/BIOS 統計オブジェクトは、各種ルーチンの実行時における絶対 CPU 使用量をトレースするためにも役立ちます。STS_set および STS_delta オペレーションを使用してプログラムの特定セクションを指定することにより、アプリケーションの個々の部分ごとに、リアルタイムのパフォーマンス統計情報を収集できます。

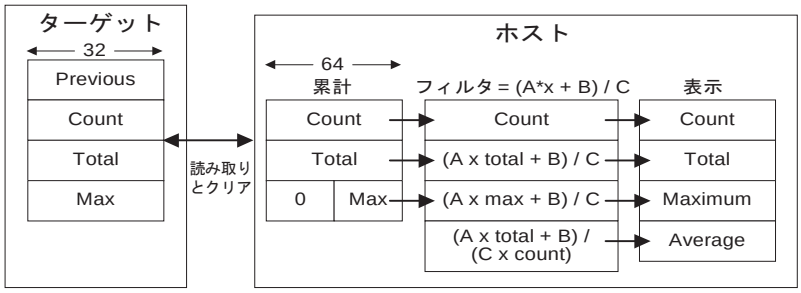
これらの統計情報は、図 3-4 に示すように「Statistics View」にリアルタイムで表示することができます。「Statistics View」にアクセスするには、「DSP/BIOS」→「Statistics View」の順に選択します。

図 3-4. 「Statistics View」パネル



統計情報は、ターゲットでは 32 ビット変数内に累積されますが、ホストでは 64 ビット変数内に累積されます。ホストは、ターゲットをポーリングしてリアルタイムに統計情報を取得すると、ターゲット上の変数をリセットします。これにより、ターゲットで必要な空間を最小限に抑え、しかも長時間のテスト実行に関する統計情報を維持することができます。「Statistics View」では、オプションとして、図 3-5 に示すように、表示の前にデータを算術フィルタに掛けることができます。

図 3-5.　ターゲット / ホスト変数の累算



ホストはターゲット上の値をクリアするので、ターゲット上の値が 0 に折り返してデータが失われる危険を防止でき、しかもはるかに多くの値を表示することができます。STS データのポーリングがディセーブルにされているか、またはポーリング・レートが非常に小さいときには、STS データが折り返している可能性があり、結果として誤った情報が提供されることがあります。

ターゲット上の値はホストが自動的にクリアしますが、ホストに保存される 64 ビット・オブジェクトはユーザがクリアすることができます。そのためには、「STS Data」ウィンドウ上で右クリックし、ショートカット・メニューから「Clear」を選択します。

ホストによる読み取りとクリアは割り込みをディセーブルにした状態で行われるので、どのスレッドがどの STS オブジェクトを更新しても信頼性が失われることはありません。たとえば、ある HWI 関数が STS オブジェクトに対する STS_add を呼び出しても、STS のどのフィールドのデータも失われません。

この計測プロセスは、ターゲット・プログラムへの妨害を最小限に抑えることができます。STS_add を 1 回呼び出すために必要な命令数は、C5000 プラットフォームでは約 20 個、C6000 プラットフォームでは 18 個です。同様に、STS オブジェクトが使用するデータ・メモリの量は、C5000 プラットフォームでは 8 ワード、そして C6000 プラットフォームでは 4 ワードにすぎません。データのフィルタリング、書式化、および平均値計算はホストで行われます。

統計情報を取得するためのポーリング・レートは、「RTA Control Panel」の「Property Page」を使用して制御することができます。ポーリング・レートを 0 にセットした場合は、ユーザが「Statistics View」ウインドウで右クリックしポップアップ・メニューから「Refresh Window」を選択しない限り、ホストは STS オブジェクトに関する情報を取得するためのターゲットのポーリングを行いません。

3.3.3.1 可変値に関する統計情報

STS オブジェクトは、32 ビット・データ値の時系列に沿った統計情報を累積するために使用できます。

たとえば、オーディオ・データの i 番目のフレーム上でアルゴリズムにより検出されるピッチを P_i とします。STS オブジェクトは、時系列 $\{P_i\}$ に関する要約情報を保存します。次の部分コードには、STS オブジェクトによりトレースされる一連の値における現在のピッチ値が含まれています。

```
pitch = `do pitch detection`
STS_add(&stsObj, pitch);
```

「Statistics View」には、系列内の値の数、最大値、系列内のすべての値の合計、および平均値が表示されます。

3.3.3.2 時間枠に関する統計情報

どのようなリアルタイム・システムにも、重要な時間枠がいくつかあります。時間枠は連続した時刻値間の差を表すので、STS はこれらの測定値に対する明示的なサポートを提供します。

たとえば、アルゴリズムがデータの i 番目のフレームを処理するために要した時間を T_i とします。STS オブジェクトは、時系列 $\{T_i\}$ に関する要約情報を保存します。次の部分コードは、CLK_gethtime（高分解能時間）、STS_set、および STS_delta を使用して、アルゴリズムを実行するための所要時間に関する統計情報をトレースする方法を示しています。

```
STS_set(&stsObj, CLK_gethtime());
    `do algorithm`
STS_delta(&stsObj, CLK_gethtime());
```

STS_set は、CLK_gethtime の値を STS オブジェクト内の「Previous value」（前の値）フィールドの内容（set 値）としてセーブします。STS_delta は、渡された新しい値からこの set 値を引きます。その結果、アルゴリズムが開始される前に記録された時刻と完了後に記録された時刻の差、つまりアルゴリズムの実行に要した時間（T_i）が算出されます。次に STS_delta は STS_add を起動し、この結果を、トレース対象とする「Previous value」フィールドの新しい値として渡します。

ホストは、アルゴリズムの実行回数、アルゴリズムの実行に要した最大時間、アルゴリズムの合計実行時間および平均実行時間を表示することができます。

set 値は、STS オブジェクトの 4 番目のコンポーネントです。これは絶対値ではなく、値の差から成るデータ系列の統計解析をサポートする目的で提供されているフィールドです。

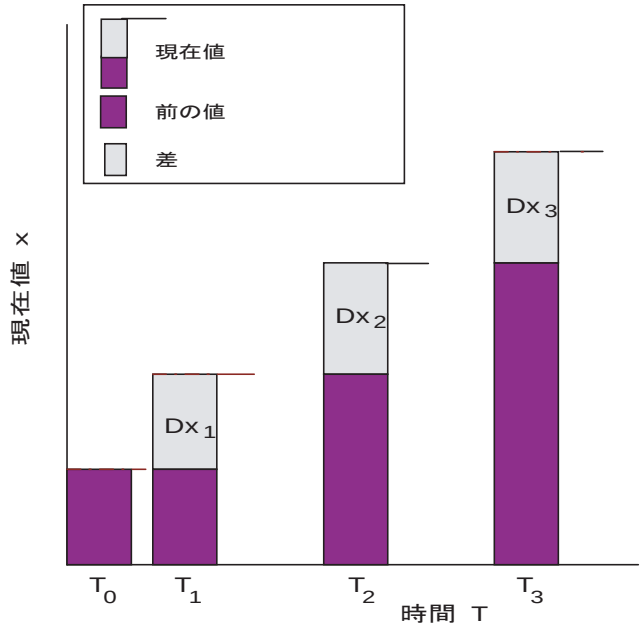
3.3.3.3 値の差に関する統計情報

STS_set および STS_delta は、STS オブジェクト内の「Previous value」フィールドの内容を更新します。呼び出しシーケンスに従って、特定の値の差、または前回の STS 更新以降の値の差を測定できます。例 3-1 に、特定の値の差に関する情報を収集するためのコードを示します。図 3-6 に、基礎値と現在値の差を示します。

例 3-1. 値の差に関する情報の収集

```
STS_set(&sts, targetValue);    /* T0 */
"processing"
STS_delta(&sts, currentValue); /* T1 */
"processing"
STS_delta(&sts, currentValue); /* T2 */
"processing"
STS_delta(&sts, currentValue); /* T3 */
"processing"
```

図 3-6. 1 つの STS_set を基準とした現在値の差

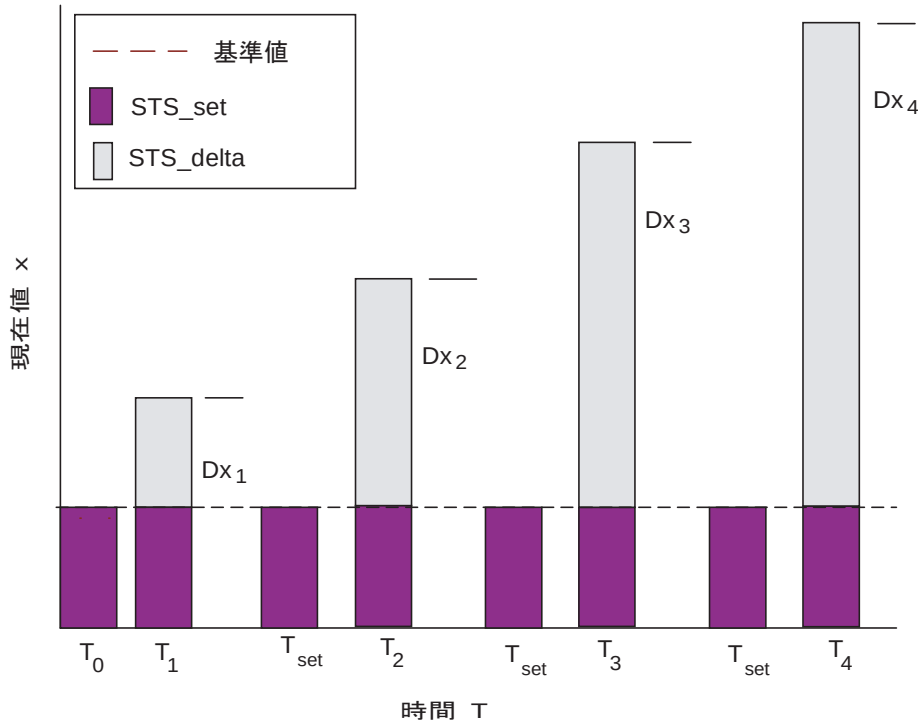


例 3-2 に、基礎値を基準とした値の差に関する情報を収集するコードを示します。図 3-7 に、基礎値と現在値の差を示します。

例 3-2. 基礎値との差に関する情報の収集

```
STS_set(&sts, baseValue);
"processing"
STS_delta(&sts, currentValue);
STS_set(&sts, baseValue); "processing"
STS_delta(&sts, currentValue);
STS_set(&sts, baseValue);
"processing"
STS_delta(&sts, currentValue);
STS_set(&sts, baseValue);
"processing"
```

図 3-7. 基礎値と現在値の差



統計オブジェクト、およびそのパラメータについては、オンライン・ヘルプに説明があります。STS モジュールの API コールについては、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「STS Module」を参照してください。

3.3.4 トレース・マネージャ（TRC モジュール）

TRC モジュールを使用すると、アプリケーションが、リアルタイムでの解析データの取得をイネーブルにしたりディセーブルにしたりできるようになります。たとえば、ターゲットで、アプリケーションの動作に異常が認められたときに、TRC モジュールを使用することによってデータの取得を停止または開始することができます。

データ収集を制御することは重要なオペレーションです。これにより、計測がプログラムの動作に及ぼす影響を抑制し、必要な情報が LOG オブジェクトと STS オブジェクトに確実に含まれることになり、実行時にイベントやデータ値の記録を開始または停止することができるからです。

たとえば、あるイベントが発生したときに計測をイネーブルにすれば、固定ログを使用して、そのログをイネーブルにしてから発生した最初の n 個のイベントを保存することができます。あるイベントが発生したときにトレースをディセーブルにすれば、循環ログを使用して、そのログをディセーブルにする前の最後の n 個のイベントを保存することができます。

3.3.4.1 明示計測の制御

明示計測を制御するには、次の部分コードに示すように TRC モジュールを使用します。

```
if (TRC_query(TRC_USER0) == 0) {
    `LOG or STS operation`
}
```

注：

TRC_query は、渡されたマスク内のトレース・タイプがすべてイネーブルになっていれば 0 を返し、マスク内のトレース・タイプのいずれかがディセーブルになっていれば 0 以外の値を返します。

この部分コードのオーバーヘッドは、テスト対象のビットがセットされていない場合は、ほんの数個の命令サイクルに過ぎません。アプリケーションに、テストおよび関連の計測呼び出しに必要な余分なプログラム・サイズを受け入れる余裕があれば、このコードを実動アプリケーションに組み込んでおくことにより開発プロセスが簡素化されフィールド診断も可能になるので、大変便利です。実際に、このモデルは DSP/BIOS 計測カーネルの中で使用されています。

3.3.4.2 暗黙計測の制御

TRC モジュールは、一組のトレース・ビットを使用して、ログおよび統計オブジェクトを用いた暗黙計測データのリアルタイムのキャプチャを制御します。効率を高めるために、ターゲットでは、トレースがイネーブルにされている場合以外はログまたは統計情報を保存しません（LOG_printf または LOG_event を使用して明示的に書き込まれたメッセージ、および STS_add または STS_delta を使用して追加された統計情報については、トレースをイネーブルにする必要はありません）。

DSP/BIOS は、特定のトレース・ビットを参照するために、図 3-2 に示す定数を定義しています。ターゲット・アプリケーションは、トレース・ビットを使用してシステム情報の収集の開始時と停止時を制御することができます。この機能は、特定の 1 つまたは一組のイベントに関する情報をキャプチャする場合に便利です。

デフォルトでは、すべての TRC 定数はイネーブルです。しかし、TRC_GBLHOST または TRC_GBLTARG のいずれかの定数がディセーブルの場合は、TRC_query はゼロ以外の値を返します。これは、これらのビットがセットされていない限り、トレースは行われないからです。

表 3-2. TRC 定数

定数	トレースのイネーブル / ディセーブル	デフォルト
TRC_LOGCLK	低分解能クロック割り込みを記録します。	オン
TRC_LOGPRD	システム・ティックおよび周期関数の開始を記録します。	オン
TRC_LOGSWI	ソフトウェア割り込み関数のポスト、起動、完了を記録します。	オン
TRC_LOGTSK	タスクがレディ状態になったとき、開始されたとき、ブロックされたとき、実行を再開したとき、および終了したときにイベントを記録します。この定数はセマフォ・ポストも記録します。	オン
TRC_STSHWI	HWI 内の監視対象のレジスタ値に関する統計情報を収集します。	オン
TRC_STSPIP	データ・パイプから読み出したフレーム、またはそこに書き込まれたフレームの数をカウントします。	オン
TRC_STSPRD	周期関数の実行中に経過したティック数に関する統計情報を収集します。	オン
TRC_STSSWI	ソフトウェア割り込みのポストから完了までの命令サイクル数または経過時間に関する統計情報を収集します。	オン
TRC_STSTSK	タスクが実行可能な状態になってから、TSK_deltatime() に対する呼び出しが発行されるまでの、TSK 実行の長さに関する統計情報を収集します。これは、タイマ割り込み単位数または CLK ティック数で測定されます。	オン
TRC_USER0 および TRC_USER1	一連の明示計測オペレーションをイネーブルまたはディセーブルにします。TRC_query を使用すると、これらのビットの設定をチェックし、その結果に基づいて呼び出しを実行または省略することができます。DSP/BIOS は、これらのビットを使用しないし、セットもしません。	オン
TRC_GBLHOST	イネーブルにされているすべてのタイプのトレースの収集を、同時に開始または停止します。暗黙計測を実行するには、このビットがセットされている必要があります。これは、異なるタイプのイベントを相互に関連付ける場合に重要です。通常このビットは、ホストで「RTA Control Panel」を使用して実行時にセットします。	オン
TRC_GBLTARG	暗黙計測を制御します。暗黙計測を実行するには、このビットもセットされている必要があります。また、このビットをセットできるのはターゲット・プログラムだけです。	オン

注： タスク統計情報の更新

あるタスクで TSK_deltatime が呼び出されない場合は、「RTA Control Panel」で TSK アキュムレータがイネーブルにされていても、「Statistics View」でそのタスクの統計情報は更新されません。

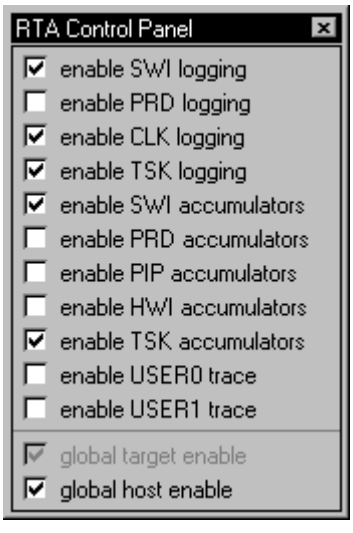
TSK の関数は、通常、他のスレッドを待っている間はブロックする無限ループを実行するので、TSK 統計情報の取り扱いは他の統計情報とは異なります。これに対して、HWI および SWI 関数はブロックなしで完了まで実行されます。このような違いがあるので、DSP/BIOS では、プログラムは TSK_settime を呼び出すことにより TSK 関数の処理ループの「始め」を識別し、TSK_deltatime を呼び出すことによりループの「終わり」を識別できるようになっています。

これらのトレース・ビットは、次のようにしてイネーブルおよびディセーブルにすることができます。

- ホストからの場合は、図 3-8 に示す「RTA Control Panel」を使用します。このパネルでは、実行時の情報収集と時間的圧迫との間のバランスを調整できます。各種の暗黙計測タイプをディセーブルにすれば一部の情報は失われますが、処理のオーバーヘッドは減少します。

  「RTA Control Panel」の「Property Page」上で右クリックすると、トレース状態情報のリフレッシュ・レートを制御することができます。リフレッシュ・レートを 0 にセットした場合は、ユーザが「RTA Control Panel」で右クリックしポップアップ・メニューから「Refresh Window」を選択しない限り、ホストはトレース状態情報を調べるためのターゲットのポーリングを行いません。最初は、デフォルトで「RTA Control Panel」のすべてのボックスにチェックマークが付いています。

図 3-8. 「RTA Control Panel」ダイアログ・ボックス



- ターゲット・コードからの場合は、TRC_enable および TRC_disable オペレーションをそれぞれ使用して、トレース・ビットをイネーブルにしたりディセーブルにしたりします。たとえば、次の C コードは、ソフトウェア割り込みと周期関数に関するログ情報のトレースをディセーブルにします。

  ```
  TRC_disable(TRC_LOGSWI | TRC_LOGPRD);
  ```

  たとえば、夜間実行中に、ある特定の状況が発生するかどうか知りたいとします。この場合プログラムは、その状況が生じたときに次の文を実行してすべてのトレースをオフにし、その時点の計測情報を保存できます。

  ```
  TRC_disable(TRC_GBLTARG);
  ```

ターゲット・プログラムがトレース・ビットに対して行った変更は、すべて「RTA Control Panel」に表示されます。たとえば、あるイベントが発生したときにターゲット・プログラムが情報のトレースをディセーブルにするように設定しておきます。この場合、ホストで「global target enable」チェック・ボックスがクリアされるまで待ってから、ログを調べるだけで済みます。

3.4 DSP/BIOS の暗黙計測

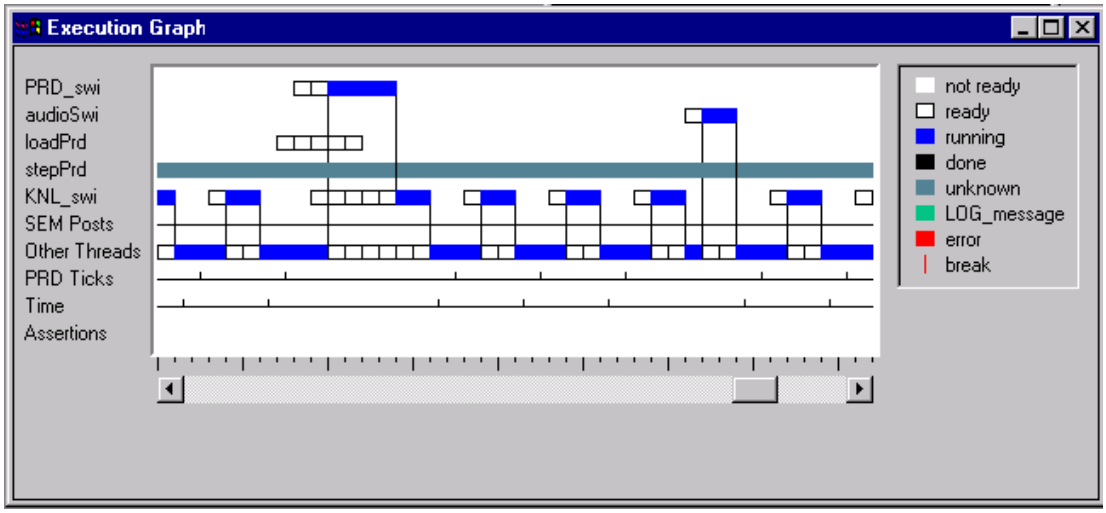
DSP/BIOS 解析ツールが「Execution Graph」、システム統計情報、および CPU の負荷を表示するために必要なすべての計測機能は、暗黙計測として DSP/BIOS プログラムの 1 つに自動的に組み込まれています。ユーザは、Code Composer Studio に含まれている「RTA Control Panel」解析ツールを使用して、DSP/BIOS 暗黙計測の個々のコンポーネントをイネーブルにできます。その方法については、3.3.4.2 項「暗黙計測の制御」（3-16 ページ）を参照してください。

DSP/BIOS 計測は効率的に設計されており、すべての暗黙計測がイネーブルにされているときでも、一般的なアプリケーションの場合の CPU の負荷の増加は 1% 未満です。計測パフォーマンスの詳細については、3.2 節「計測のパフォーマンス」（3-3 ページ）を参照してください。

3.4.1 「Execution Graph」

「Execution Graph」は、SWI、PRD、TSK、SEM、および CLK の処理に関する情報を表示するための特殊なグラフです。これらのオブジェクト・タイプのロギングは、実行時にホストで TRC モジュール API または「RTA Control Panel」を使用して、個別にイネーブルまたはディセーブルにすることができます。「Execution Graph」上のセマフォ・ポストを制御するには、TSK ロギングをイネーブルまたはディセーブルにします。「ExecutionGraph」ウィンドウには、図 3-9 に示すように「Execution Graph」情報が各オブジェクトの動作を示すグラフとして表示されます。

図 3-9.　「Execution Graph」ウィンドウ



「Execution Graph」には、時間間隔の測定値を表すものとして CLK イベントと PRD イベントが示されます。「Execution Graph」では、個々のログ・イベントのタイムスタンプを取るのではなく、単に CLK イベントを他のシステム・イベントと一緒に記録するだけです。タイムスタンプ・オペレーションを行うとタイムスタンプを取得するための時間と余分なログ空間が必要になるので、デバイスに要求される性能が増加します。したがって、「Execution Graph」の時間目盛りはリニアにはなりません。

「Execution Graph」には、SWI、TSK、SEM、PRD、および CLK イベントに加えて、その他の情報がグラフ形式で表示されます。「Assertions」は、リアルタイム・デッドラインに違反したこと、または無効な状態が検出されたこと（システム・ログが壊れているか、またはターゲットが不法操作を行ったため）を表示します。ユーザのアプリケーションが行った LOG_message 呼び出しの場合は、LOG_message 状態（緑で表される）が「Assertions」のトレース行に表示されます。内部ログ呼び出しにより生成されたエラーは、「Assertions」のトレース行に赤で表示されます。「Assertions」のトレース上の赤のボックスは、システム・ログから収集された情報の区切りを示します。

DSP/BIOS プログラムの実行に関連した「Execution Graph」の情報を解釈する方法は、4.1.5 項「優先権の譲渡と優先実行」（4-8 ページ）を参照してください。

3.4.2 CPU の負荷

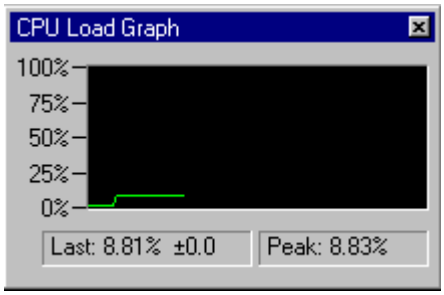
CPU の負荷は、CPU がアプリケーション作業を行うために消費する命令サイクルのパーセンテージで、つまり CPU が次の処理を行うために費やす合計時間のパーセンテージです。

- ハードウェア割り込み、ソフトウェア割り込み、または周期関数を実行する。
- ホストとの間の入出力（I/O）処理を実行する。
- 何らかのユーザ・ルーチンを実行する。
- 省電力モードまたはハードウェア・アイドル・モード状態（'C55x のみ）。

CPU が上記のどの作業も行っていないときは、アイドル状態と見なされます。

CPU が PWRM モジュールを通じて 'C55x でサポートされている省電力モード中にアイドル状態になっていても、DSP/BIOS のアイドル・ループは実行されません。したがって、CPU の負荷は計算されず、100% と表示されます。

「CPU Load Graph」ウィンドウを表示するには、「DSP/BIOS」→「CPU Load Graph」の順に選択します。



CPU のすべての動作は、作業時間とアイドル時間のどちらかに分類されます。時間間隔 T における CPU の負荷を測定するには、その時間内にアプリケーション作業を行うために消費された時間（t_w）と、アイドル状態だった時間（t_i）を知る必要があります。これに基づき、次のようにして CPU の負荷を計算できます。

$$CPUload = \frac{t_w}{T} \times 100$$

CPU は、常に作業を行っているかアイドルかのどちらかの状態なので、次の式が成り立ちます。

$$T = t_w + t_i$$

この式は、次のように書き換えることができます。

$$CPUload = \frac{t_w}{t_w + t_i} \times 100$$

CPU の負荷は、次のように命令サイクル数を使用して表すこともできます。

$$CPUload = \frac{c_w}{c_w + c_i} \times 100$$

3.4.2.1 CPU の負荷を測定する方法

DSP/BIOS アプリケーションでは、CPU が作業状態になるのは次のいずれかが行われているときです。

- ハードウェア割り込みに対するサービスの実行中
- ソフトウェア割り込みおよび周期関数の実行中
- タスク関数の実行中
- アイドル・ループからのユーザ関数の実行中
- HST チャネルがホストにデータを転送中
- リアルタイム解析データを DSP/BIOS 解析ツールにアップロード中

これらの作業のいずれも行っていないとき、CPU はアイドル・ループに入り、IDL_cpuLoad 関数を実行し、他の DSP/BIOS IDL オブジェクトを呼び出しています。つまり、DSP/BIOS アプリケーションにおける CPU アイドル時間とは、CPU が例 3-3 に示すルーチンを実行するために消費している時間です。

時間間隔 T の間の DSP/BIOS アプリケーションにおける CPU の負荷を測定するには、図 3-3 に示すループに入っていた時間と、アプリケーション作業に消費した時間が分かっていれば十分です。

例 3-3. アイドル・ループ

```
'Idle_loop:
    Perform IDL_cpuLoad
    Perform all other IDL functions (user/system functions)
    Goto IDL_loop'
```

時間を T とし、MIPS（百万命令 / 秒）を M とした場合、CPU は M x T 個の命令サイクルを実行することになります。これらの命令サイクルのうち、c_w 個がアプリケーション作業を行うために費やされます。残りは、例 3-3 に示したアイドル・ループを実行するために費やされます。このループを 1 回実行するために必要な命令サイクルの数を l_1 とすると、ループを実行するために費やされる命令サイクルの合計数は $N \times l_1$ になります（ここで、N は時間 T の間にループが反復実行される回数です）。したがって、合計命令サイクル数は、作業命令サイクル数とアイドル命令サイクル数の和です。

$$MT = c_w + Nl_1$$

この式から、c_w は次のように表すことができます。

$$c_w = MT - Nl_1$$

3.4.2.2 アプリケーションの CPU の負荷を計算する方法

前述の式を使用すると、DSP/BIOS アプリケーションにおける CPU の負荷は、次のようにして計算することができます。

$$\text{CPUload} = \frac{c_w}{MT} \times 100 = \frac{MT - NI_l}{MT} \times 100 = \left(1 - \frac{NI_l}{MT}\right) \times 100$$

CPU の負荷を計算するには、l_1 の値、および CPU の負荷の測定対象時間 T の間の N の値が分かっていなければなりません。

DSP/BIOS アイドル・ループの中の IDL_cpuLoad オブジェクトは、STS オブジェクト IDL_busyObj を更新します。このオブジェクトには、IDL_loop の実行回数と、DSP/BIOS 高分解能クロック（4.9 節「タイマ、割り込み、およびシステム・クロック」（4-71 ページ）を参照）に保持されている時間が記録されています。ホストは、この情報を使用して、上記の式に従って CPU の負荷を計算します。

ホストは、「RTA Control Panel」の「Property Page」で設定されているポーリング・レートで、STS オブジェクトをアップロードします。IDL_busyObj に含まれている情報を使用して、CPU の負荷が計算されます。IDL_busyObj カウントは、N（アイドル・ループが実行された回数）の測定値を提供します。CPU の負荷の計算には、IDL_busyObj の最大値は使用されません。IDL_busyObj の合計値には、低分解能クロックの単位で T の値が示されます。

CPU の負荷を計算するには、さらに l_1（アイドル・ループに費やされた命令サイクル数）が分かっていなければなりません。アイドル関数マネージャで「Auto calculate idle loop instruction count」ボックスが選択されている場合、DSP/BIOS は、初期化の時点で BIOS_init に基づいて l_1 を計算します。

ホストは N、T、l_1、および CPU MIPS の値を使用して、次のように CPU の負荷を計算します。

$$\text{CPUload} = \left[1 - \frac{Nl_1}{MT}\right]100$$

3.4.3 ハードウェア割り込みカウントと最大スタック深度

個々の HWI 関数がトリガされた回数をトレースすることができます。そのためには、HWI オブジェクトに monitor パラメータを設定してスタック・ポインタを監視します。図 3-10 および 3-11 に示すように、監視対象の各ハードウェア ISR について STS オブジェクトが 1 つずつ自動的に作成されます。

図 3-10. スタック・ポインタを監視する方法（C5000 プラットフォーム）

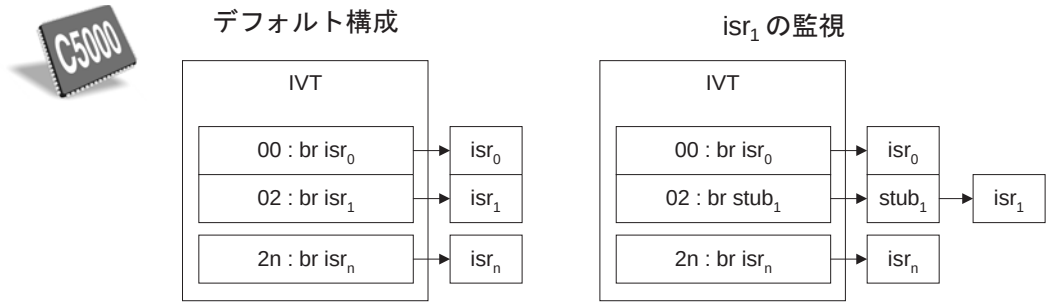
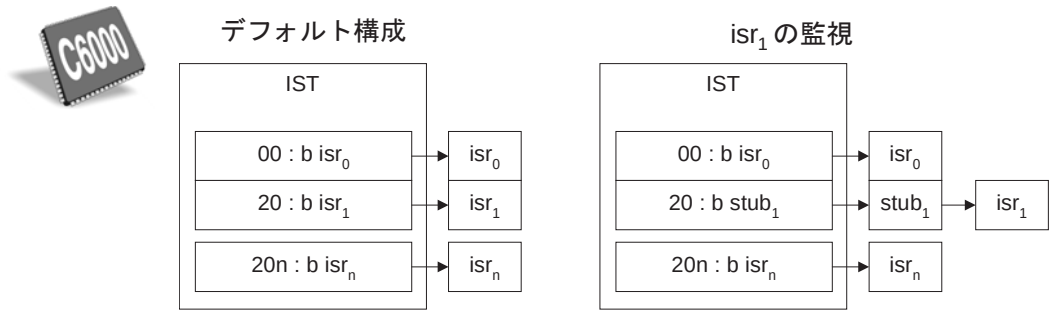


図 3-11. スタック・ポインタを監視する方法（C6000 プラットフォーム）



監視されないハードウェア割り込みについては、オーバーヘッドはまったく生じません。つまり、制御は HWI 関数に直接渡されます。監視対象の割り込みの場合は、まず構成が生成するスタブ関数に制御が渡されます。この関数は選択されたデータ位置を読み取り、その値を選択された STS オペレーションに渡し、最後に HWI 関数に分岐します。

HWI 関数の監視が行われるようにするには、「RTA Control Panel」内の「enable HWI accumulations」チェックボックスを選択する必要があります。このタイプのトレースがイネーブルになっていない場合、スタブ関数は、STS オブジェクトを更新せずに HWI 関数に分岐します。

STS オブジェクトの「Count」フィールドには、割り込みがトリガされた回数が記録されます。スタック・ポインタを監視している場合、最大値は、割り込みが発生したときのシステム・スタックの最上部の最大位置を示します。これは、アプリケーションが必要とするシステム・スタック・サイズを判別するために役立ちます。スタックの最大深度を判別する手順は、次のとおりです（図 3-12 を参照）。

1) Tconf スクリプト内で、HWI オブジェクトの「monitor」フィールドを「Stack Pointer」にセットします。また、「operation」フィールドを STS_add（–*addr）に変更します。

   これらの変更をすると、STS オブジェクトの「maximum」フィールドに、スタック・ポインタの最小値が表示されます。スタックが成長するとともにメモリは減少するので、これがスタックの最上部になります。

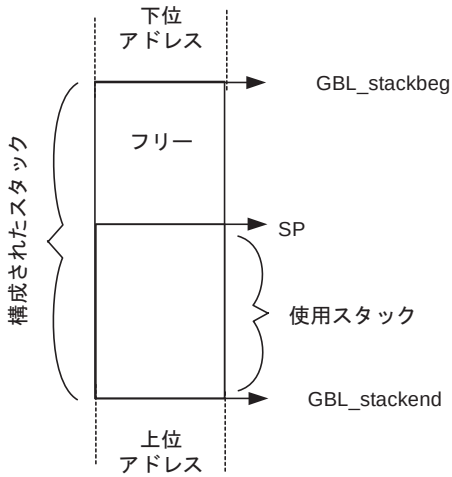
2) プログラムをリンクし、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の第 2 章「Utility Programs」で説明されている nmti プログラムを使用して、アプリケーション・スタックの終わりのアドレスを見つけます。あるいは、Code Composer Studio で「Memory」ウィンドウまたはマップ・ファイルを使用して GBL_stackend シンボルにより参照されているアドレスを見つけることで、アプリケーション・スタックの底のアドレスを見つけることもできます （このシンボルはスタックの最上部を参照します）。

3) プログラムを実行し、「Statistics View」ウィンドウに、この HWI 関数のスタック・ポインタを監視する STS オブジェクトを表示します。

4) システム・スタックの底からスタック・ポインタの最小値（STS オブジェクトの最大値フィールド）を引いて、スタックの最大深度を求めます。

「Kernel Object View」に、すべてのターゲットに関する情報が表示されます （3.5 節「Kernel Object View」を参照）。

図 3-12.　使用されているスタック深度を計算する方法



使用スタック深度 = {GBL_stackend - min(SP)}

STS_add(–*addr) = min(SP)

3.4.4　変数を監視する方法

ハードウェア割り込みの発生回数をカウントし、スタック・ポインタを監視することに加えて、ハードウェア割り込みがトリガされるたびに任意のレジスタまたはデータ値を監視することができます。

この暗黙計測は、どの HWI オブジェクトについてもイネーブルにできます。このような監視は、デフォルトではイネーブルになっていません。つまり、構成でこのタイプの計測をイネーブルにしない限り、割り込み処理のパフォーマンスに影響はありません。統計オブジェクトは、ハードウェア割り込み処理が開始されるたびに更新されます。このような統計オブジェクトを更新すると、監視対象の各割り込みについて、1 回の割り込み当たり 20 ～ 30 命令が実行されます。

暗黙的な HWI 計測をイネーブルにする手順は、次のとおりです。

1) 任意の HWI オブジェクトに関するプロパティ・ウィンドウを開き、「monitor」フィールドで監視対象のレジスタを選択します。

   表 3-3 に示す変数は、どれでも監視できます。また、何も監視しないという選択もできます。監視する変数を選択すると、構成は、その変数に関する統計情報を保存するために STS オブジェクトを自動的に作成します。

表 3-3. HWI を使って監視できる変数

C54x プラットフォーム 	C55x プラットフォーム 	C6000 プラットフォーム 	C28x プラットフォーム 
データ値	データ値	データ値	データ値
システム・スタックの最上部		システム・スタックの最上部	
スタック・ポインタ	スタック・ポインタ	スタック・ポインタ	スタック・ポインタ
汎用レジスタ： ag ar6 imr ah ar7 pmst al bg rea ar0 bh rsa ar1 bk st0 ar2 bl st1 ar3 brc t ar4 ifr tim ar5 trn	汎用レジスタ： ac0 rea1 trn0 ac1 rptc trn1 ac2 rsa0 xar0 ac3 rsa1 xar1 brc0 st0 xar2 brc1 st1 xar3 ifr0 st2 xar4 ifr1 st3 xar5 imr0 t0 xar6 imr1 t1 xar7 reta t2 xcdp rea0 t3 xdp	汎用レジスタ： a0 a12 b6 a1 a13 b7 a2 a14 b8 a3 a15 b9 a4 **a16- a31** b10 a5 **（C64x のみ）** b11 a6 b12 a7 b0 b13 a8 b1 b14 a9 b2 b1 a10 b3 **b16- b31** a11 b4 **（C64x のみ）** b5	汎用レジスタ： ah ph xar0 al pl xar1 idp st0 xar2 ifr st1 xar3 ier t xar4 tl xar5 xar6 xar7

2) operation パラメータを、この値に対して実行したい STS オペレーションにセットします。

   選択した変数に保存されている値に対して、表 3-4 に示すいずれかのオペレーションを行うことができます。どのオペレーションの場合も、「count」フィールドにはこのハードウェア割り込みが実行された回数が保存されます（図 3-5 を参照）。max および total の値は、ターゲット上の STS オブジェクトに保存されます。average はホストで計算されます。

表 3-4. STS オペレーションとその結果

STS オペレーション	結果
STS_add(*addr)	データ値またはレジスタ値の最大および合計を保存します。
STS_delta(*addr)	データ値またはレジスタ値を、STS オブジェクトの prev プロパティ（または STS_set により定数として設定されている値）と比較して、最大と合計の差を保存します。
STS_add(-*addr)	データ値またはレジスタ値を -1 倍し、最大と合計を保存します。その結果、最大として保存された値は -1 倍された最小値です。合計と平均は、-1 倍された合計値と平均値です。
STS_delta(-*addr)	データ値またはレジスタ値を -1 倍し、データ値またはレジスタ値を STS オブジェクトの prev プロパティ（または STS_set によりセットされている値）と比較します。最大と合計の差を保存します。その結果、最大として保存された値は -1 倍された最小差です。
STS_add(|*addr|)	データ値またはレジスタ値の絶対値をとり、最大および合計を保存します。その結果、最大として保存された値は最大の負または正の値です。平均は絶対値の平均です。
STS_delta(|*addr|)	レジスタ値またはデータ値の絶対値を、STS オブジェクトの prev プロパティ（または STS_set によりセットされている値）と比較します。最大と合計の差を保存します。その結果、最大として保存された値は最大の負または正の差で、平均は指定した値からの平均差です。

3) 対応する STS オブジェクトのプロパティをセットすることにより、ホスト上でこの STS オブジェクトの値をフィルタリングすることもできます。

たとえば、システム・スタックの最上部を監視して、アプリケーションが割り振られたスタック・サイズを超過するかどうか確認したい場合があります。システム・スタックの最上部は、プログラムのロード時に、C5000 プラットフォームでは 0xBEEF に、C6000 プラットフォームでは 0xC0FFEE にそれぞれ初期化されます。この値が変化したときは、アプリケーションが割り当てられたスタックを超過しているか、何らかのエラーが原因でそのアプリケーションがアプリケーションのスタックを上書きしています。

次の手順は、割り当てられたスタック・サイズを超過しているかどうかを監視するための方法の 1 つです。

1) 構成時に、定常的に発生する HWI 関数に対する暗黙計測をイネーブルにします。その HWI オブジェクトの「monitor」プロパティを「Top of SW Stack」に変更し、オペレーションには STS_delta(*addr) を指定します。

2) 対応する STS オブジェクトの「prev」プロパティを、C5000 および C2800 プラットフォームでは 0xBEEF に、C6000 プラットフォームでは 0xC0FFEE にそれぞれセットします。

3) Code Composer Studio でプログラムをロードし、「Statistics View」を使用して、この HWI 関数のスタック・ポインタを監視する STS オブジェクトを見ます。

4) プログラムを実行します。スタックの最上部の値の変化は、対応する STS オブジェクト内の合計値（または最大値）がゼロ以外の値であることを示します。

3.4.5 割り込みレイテンシ

割り込みレイテンシは、割り込みのトリガの時点から HWI の最初の命令が実行されるまでの最大時間です。タイマ割り込みについて割り込みレイテンシを測定するには、ご使用のプラットフォームに応じて次のいずれかのステップを行います。

1) TIM レジスタを監視するように、HWI_TINT オブジェクトを構成します。
2) operation パラメータを STS_add(-*addr) にセットします。
3) HWI_TINT_STS オブジェクトの Host Operation パラメータを A*x + B にセットします。A を 1 に、B を PRD レジスタ（グローバル CLK プロパティ・リストに示されている）の値にそれぞれセットします。

注 :

この時点で、DSP/BIOS を使用して C5500 の割り込みレイテンシを計算することはできません。これは、C55x タイマ・アクセスがデータ空間の外部にあるからです。

1) データ値を監視するように、CLK マネージャの「CPU Interrupt」プロパティに指定されている HWI オブジェクトを構成します。
2) addr パラメータを、CLK マネージャが使用するオンデバイス・タイマ用のタイマ・カウンタ・レジスタのアドレスにセットします。
3) 型を符号なしにセットします。
4) operation パラメータを STS_add(*addr) にセットします。
5) 対応する STS オブジェクト HWI_INT14_STS の Host Operation パラメータを、A * X+ B にセットします。A を 4 に、B を 0 にセットします。

1) TIM レジスタを監視するように、HWI_TINT オブジェクトを構成します。
2) operation パラメータを STS_add(*addr) にセットします。
3) HWI_TINT_STS オブジェクトの Host Operation パラメータを A*x + B にセットします。A を -1 に、B を PRD レジスタの値にそれぞれセットします。

STS オブジェクト HWI_TINT_STS（C5000）または HWI_INT14_STS（C6000）に、タイマ割り込みがトリガされた時点からタイマ・カウント・レジスタの読み取りが可能になった時点までの最大時間（命令サイクルの数）が表示されます。これが、タイマ割り込みについて生じた割り込みレイテンシです。システム内の割り込みレイテンシは、少なくともこの値と同じ大きさになります。

3.5 Kernel Object View

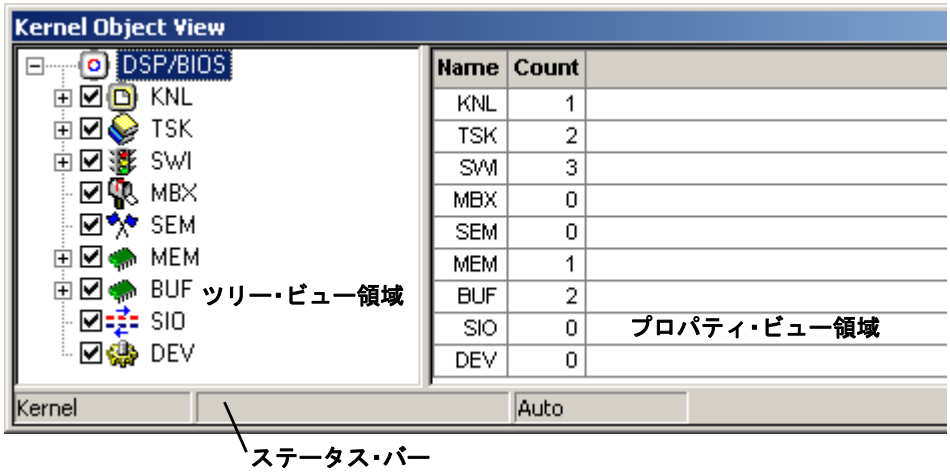
Kernel Object View ツールを使用すると、ターゲット上で実行中の DSP/BIOS オブジェクトの現在の構成、状態、およびステータスを表示することができます。

Code Composer Studio で「Kernel Object View」を開くには、表示されている「ツールバー」アイコンをクリックするか、「DSP/BIOS」→「Kernel/Object View」の順に選択します。必要に応じて、複数の「Kernel Object View」を開くこともできます。「Kernel Object View」には、3 つの領域があります。

- **ツリー・ビュー領域 :**「Kernel Object View」の左側には、アプリケーションの DSP/BIOS オブジェクトのツリー・ビューが表示されます。
- **プロパティ・ビュー領域 :** 右側には、選択したオブジェクトまたはオブジェクトの各種プロパティが表形式で表示されます。
- **ステータス・バー :**「Kernel Object View」の下部には、プログラムが実行している部分（カーネルまたはアプリケーション）を示すステータス・バーが表示されます。カーネル部分は、ステップイン実行できない DSP/BIOS コードです。アプリケーション部分は、プログラム・コードです。

ステータス・バーには、TSK または SWI のいずれのスレッドが現在実行されているのかも表示されます。

図 3-13. 最上位のオブジェクトのリストとオブジェクトのカウントを示す「Kernel Object View」

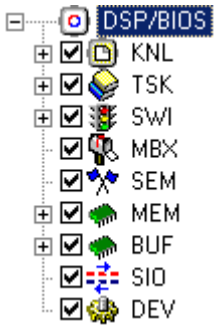


注 : 「Kernel Object View」のデータ更新

他の DSP/BIOS 解析ツールと違い、「Kernel Object View」のデータは、ターゲットが（ブレークポイントなどで）停止したとき、または「Kernel Object View」を右クリックしてポップアップ・メニューから「Refresh」を選択したときにのみ更新されます。ターゲットを一時的に停止して、データ転送を行います。「Kernel Object View」のデータは、DSP/BIOS アイドル・スレッド経由では実行時に更新されません。前回の更新以降に変更されたデータは、赤で表示されます。

3.5.1 ツリー・ビューの使用方法

「Kernel Object View」の左側にあるツリー・ビューを使用して、オブジェクトの種類に合わせてデータ表示をイネーブルしたりディセーブルして、右側に表示するオブジェクトを選択することができます。表示されているオブジェクトの種類には、KNL（システム全体の情報）、TSK、SWI、MBX、SEM、MEM、BUF、SIO、および DEV があります。このリストには、静的および動的に作成されたオブジェクトの両方が含まれています。

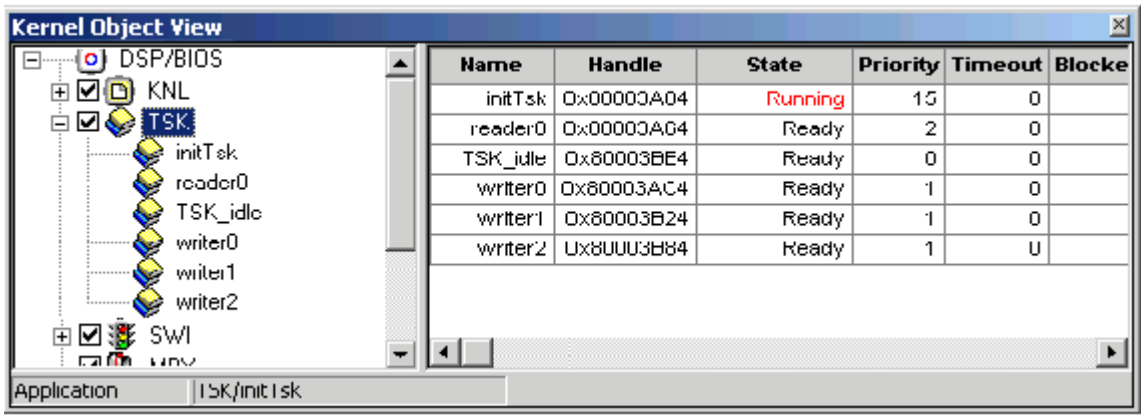


特定のタイプのオブジェクトの更新をディセーブルするには、モジュール名の横にあるボックスのチェックマークを外します。デフォルトでは、SIO と DEV のタイプは、パフォーマンスへの影響を最小限に抑えるためにディセーブルにされています。パフォーマンスをさらに改善するには、他のオブジェクト・タイプをディセーブルにします。たとえば、スレッド関連のオブジェクト（TSK、SWI、MBX、SEM）のみ、またはメモリ関連のオブジェクト（MEM、BUF）のみを対象にすることができます。

タイプのツリーを展開すると、そのタイプに該当するすべてのオブジェクトをリスト形式で表示します。これには、静的に作成されたオブジェクトや動的に作成されたオブジェクトが含まれていて、そのようなオブジェクトはデータ更新が行われたときに存在していたものです。動的に作成されたオブジェクトに名前がない場合は、自動的に生成され、不等号括弧で囲んで表示されます（たとえば、<task1>）。

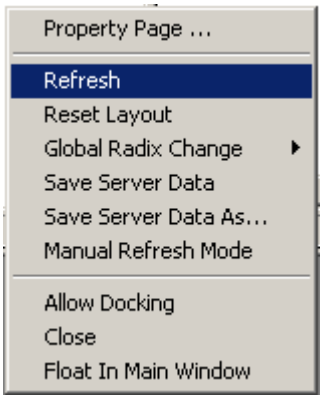
ツリー・ビューのオブジェクトをクリックすると、ウィンドウの右側にオブジェクトの各種プロパティが表示されます。Ctrl キーまたは Shift キーを使って複数のオブジェクトを選択することができます。また、オブジェクト・タイプを選択して、そのタイプに該当するすべてオブジェクトのプロパティを表示することもできます。

図 3-14. TSK のプロパティを表示する「Kernel Object View」



さまざまなタイプの複数のオブジェクトを選択した場合、「Kernel Object View」は最初に選択したオブジェクトに合わせて複数の列を表示します。選択されたその他のタイプに共通の列がある場合（例えば MBX と SEM の列は似ています）、共通の列にはそれぞれ値が表示されます。

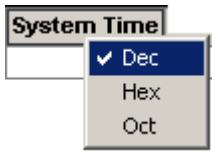
3.5.2 右クリック・メニューの使用方法



「Kernel Object View」を制御するには、空白部分を右クリックしてポップアップ・メニューを表示します。

Code Composer Studio のすべての View で用意されている View 配置コマンドに加え、このメニューには次のコマンドが含まれています。

- **Property Page:** ダイアログを開いて、「Kernel Object View」のタイトル・バーおよび「Save Server Data」コマンドを選択したときに使われるデフォルトの出力ファイルを変更することができます。
- **Refresh :** 「Kernel Object View」で表示されるデータを更新します。ターゲットを一時的に停止して、データ転送を行います。またブレークポイントなど、ターゲットを他の理由で停止した場合もデータ更新が行われます。他の DSP/BIOS 解析ツールと違い、「Kernel Object View」のデータは、DSP/BIOS アイドル・スレッド経由では実行時に更新されません。
- **Reset Layout :** 列幅、列順序、およびソート順序をデフォルトに戻します。
- **Global Radix Change :** アドレスおよびカウントなど、すべての数値を 10 進数、8 進数、16 進数のどれで表示するかを選択します。あるいは、すべての列をデフォルトにリセットすることができます。また、プロパティ・リストの数値列のヘッダを右クリックして、10 進数、8 進数、16 進数のいずれかで数値列を表示するように選択することもできます。



- **Save Server Data :** 現在のデータをファイルに保存します。デフォルトのファイルは、KOV_Data.txt ですが、右クリック・メニューの「Property Page」コマンドを使用して変更することができます。

  このファイルには、カンマで区切られたデータが含まれていて、スプレッドシートに簡単にインポートすることができます。各オブジェクト・タイプごとに、別々のデータ列には「Kernel Object View」に表示される列見出しが用意されています。

  前回の更新時にターゲットから読み出されたすべてのデータは、このファイルに含まれています。複数の「Kernel Object View」ウィンドウが開き、それぞれのウィンドウで各種オブジェクト・タイプの更新がイネーブルにされると、ファイルにはいずれかのウィンドウでイネーブルにされたオブジェクト・タイプのデータが含まれます。「Kernel Object View」の 1 つまたはすべてに対する更新がグローバルにディセーブルにされても、メモリに格納されていたデータ（およびこのようにこのファイルに保存されたデータ）は、ターゲットを停止したときに更新されます。表示されているデータに対する更新のみがディセーブルにされます。

- **Save Server Data As :** 現在のデータを保存するファイル名と場所を指定し、保存します。保存形式は、「Save Server Data」コマンドと同じです。
- **Manual Refresh Mode/Auto Refresh Mode :** 自動更新モード（デフォルト）では、プロセッサが停止すると常に、データは自動的に更新されます。手動更新モードでは、右クリック・メニューの「Refresh」コマンドを使用してデータを更新する必要があります。パフォーマンス上の理由により、コードをシングル・ステップ実行したり、複数のブレークポイントを設定して実行したりする場合、自動更新をディセーブルにすることができます。ステータス・バーには、「Kernel Object View」の更新モードが自動なのか、手動なのかが表示されます。

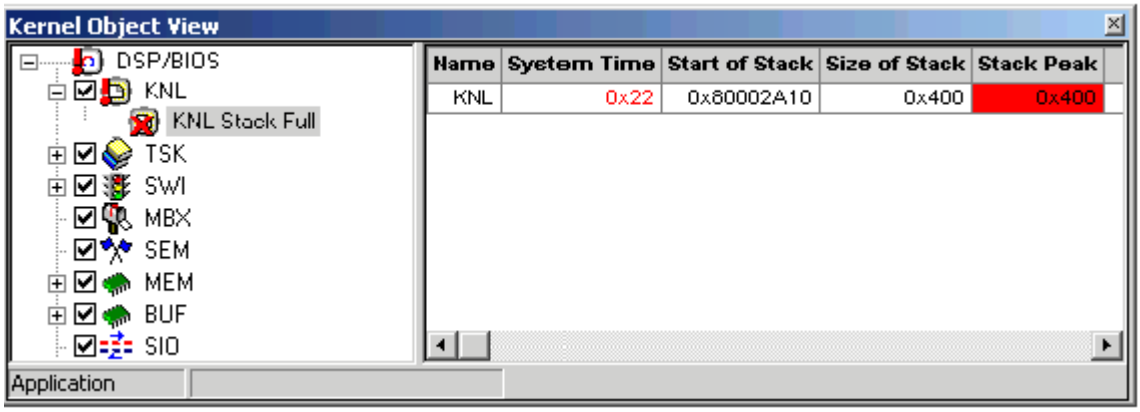
3.5.3 各種オブジェクト・タイプに合わせて表示されるプロパティ

「Kernel Object View」は、各オブジェクト・タイプに合わせて各種プロパティを表示します。各オブジェクト・タイプのプロパティは、この後の項で説明します。

前回の更新以降に変更されたデータは、赤で表示されます。

列見出しをクリックすると、プロパティ値の表をソートすることができます。列見出し間の境界をドラッグして、列幅を変更することができます。列見出しをドラッグして、列の順序を変更することできます。

任意のスタックで使用されるピーク時のスタック使用量が、スタック・サイズと同じ場合、ピーク時のサイズは赤で強調表示され、エラー・メッセージがツリー・ビューに追加されます。



3.5.3.1 カーネル

カーネル（KNL）項目は、システム全体の情報を表示します。次のようにプロパティが表示されます。

図 3-15. 「カーネル」プロパティ

Name	System Time	Start of Stack	Size of Stack	Stack Peak
KNL	0x0	0x800021C8	0x400	0x68

- **Name :** 常に KNL が表示されます。KNL タイプにはオブジェクトはありません。
- **System Time :** これはタイマ関数とタスク用アラームに使用されるクロックの現在の値です。クロックはクロック・マネージャ（CLK）で PRD_clk を構成するときに設定されます。この値が使用されるのは、タスク・マネージャ（TSK）でタスクがイネーブルにされているときだけです。タスクがディセーブルにされている場合は、「Time」フィールドはゼロのままです。
- **Start of Stack**：アプリケーションが使用するグローバル・スタックの先頭アドレス。
- **Size of Stack**：グローバル・スタックのサイズ。
- **Stack Peak**：一度に使用するグローバル・スタックのピーク時の使用量。
- **Start of SysStack**：システム・スタックの先頭アドレス （C55x のみ）。
- **Size of SysStack**：システム・スタックのサイズ （C55x のみ）。
- **SysStack Peak**：一度に使用するシステム・スタックのピーク時の使用量 （C55x のみ）。

3.5.3.2 タスク

タスク（TSK）の場合、次のようにプロパティが表示されます。

図 3-16. 「タスク」プロパティ

Name	Handle	State	Priority	Timeout	Blocked On	Start of Stack	Size of Stack	Stack Peak
TSK_idle	0x80002174	Ready	0	0		0x80001C48	0x400	0xA0
task2	0x80002114	Ready	1	0		0x80002DF8	0x400	0x64

- **Name :** タスクの名前。名前は、静的に構成されたタスクの場合はラベルから取得され、動的に作成されたタスクの場合は生成されます。ラベルは、構成時に指定した名前に一致します。
- **Handle :** ターゲット上のタスク・オブジェクト・ヘッダのアドレス。
- **State**：タスクの現在の状態を示し、「Ready」、「Running」、「Blocked」、「Terminated」のいずれかです。

- **Priority**：構成時にセットされた、または API によりセットされたタスクの優先順位。有効な優先順位は 0 ～ 15 です。
- **Timeout**：タイムアウトでブロックされている場合は、タスク・アラームが発生するクロック値。
- **Blocked On**：セマフォまたはメールボックス上でブロックされている場合は、タスクがブロックされている SEM または MBX オブジェクトの名前。たとえば、次の図には、タイムアウトが指定されてブロックされている 1 つのタスク、セマフォ上でブロックされている 3 つのタスク、および実行中の 1 つのタスクが表示されています。タスクがブロックされている項目を右クリックし、「Go To」コマンドを選択して「Kernel Object View」の該当項目に関する情報を表示することができます。

Name	Handle	State	Priority	Timeout	Blocked On	Start of Stack	Stack
tskControl	0x8000DC6C	Blocked	2	5		0x8000ED20	
tskProcess	0x8000DC0C	Blocked	2	0	SEM: 0x8001BEDC	0x8000E920	
tskRxSplit	0x8000DB4C	Blocked	2	0	SEM: semIn	0x8000E120	
tskTxJoin	0x8000DBAC	Blocked	2	0	SEM: 0x8001BE1C	0x8000E520	
TSK_idle	0x8000DCCC	Running	0	0		0x8000DD20	

- **Start of Stack**：タスク・スタックの先頭アドレス。
- **Stack Size**：タスク・スタックのサイズ。
- **Stack Peak**：一度に使用するタスク・スタックのピーク時の使用量。
- **Start of SysStack**：システム・スタックの先頭アドレス （C55x のみ）。
- **Size of SysStack**：システム・スタックのサイズ （C55x のみ）。
- **SysStack Peak**：一度に使用するシステム・スタックのピーク時の使用量 （C55x のみ）。

3.5.3.3 ソフトウェア割り込み

ソフトウェア割り込み（SWI）の場合、次のようにプロパティが表示されます。

図 3-17. 「ソフトウェア割り込み」プロパティ

Name	Handle	State	Priority	Mailbox Value	Function	arg0	arg1	Function Address
buffSwi	0x284	Inactive	0x1	0	buffAllocate	0x0	0x0	0x25B4
KNL_swi	0x258	Inactive	0x0	0	KNL_run	0x0	0x0	0x5FC0
PRD_swi	0x22C	Inactive	0x1	0	PRD_F_swi	0x0	0x0	0x5540

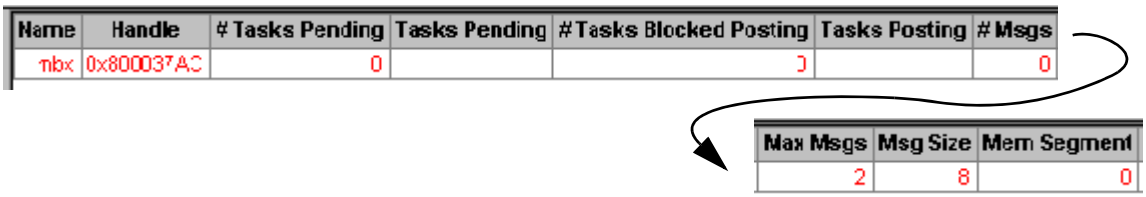
- **Name**：ソフトウェア割り込みオブジェクトの名前。名前は、静的に構成されたソフトウェア割り込みの場合はラベルから取得され、動的に作成されたソフトウェア割り込みの場合は生成されます。ラベルは、構成時に指定した名前に一致します。
- **Handle**：ターゲット上のソフトウェア割り込みオブジェクト・ヘッダのアドレス。

- **State**：ソフトウェア割り込みの現在の状態。有効な状態は「Inactive」、「Ready」、「Running」のいずれかです。
- **Priority**：構成時または作成時にセットされた、ソフトウェア割り込みの優先順位。有効な優先順位は 0 ～ 15 です。
- **Mailbox Value**：ソフトウェア割り込みの現在のメールボックス値。
- **Function**：このソフトウェア割り込みが呼び出す関数の名前。
- **arg0、arg1**：このソフトウェア割り込みが関数へ渡す引数。引数は構成時または作成時にセットされます。
- **Function Address**：ターゲット上の関数のアドレス。

3.5.3.4 メールボックス

メールボックス（MBX）の場合、次のようにプロパティが表示されます。

図 3-18. 「メールボックス」プロパティ



- **Name：**メールボックスの名前。名前は、静的に構成されたメールボックスの場合はラベルから取得され、動的に作成されたメールボックスの場合は生成されます。ラベルは、構成時に指定した名前に一致します。
- **Handle：**ターゲット上のメールボックス・オブジェクト・ヘッダのアドレス。
- **# Tasks Pending**：現在ブロックされていて、このメールボックスからメッセージの読み取りを待っているタスクの数。
- **Tasks Pending**：現在ブロックされていて、このメールボックスからメッセージの読み取りを待っているタスク名のプルダウン・リスト。選択したタスクを右クリックし、「Go To」コマンドを選択して「Kernel Object View」の該当タスクを表示することができます。
- **# Tasks Blocked Posting**：現在ブロックされていて、このメールボックスへのメッセージの書き込みを待っているタスクの数です。
- **Tasks Posting**：現在ブロックされていて、このメールボックスへのメッセージの書き込みを待っているタスク名のプルダウン・リスト。選択したタスクを右クリックし、「Go To」コマンドを選択して「Kernel Object View」の該当タスクを表示することができます。
- **# Msgs**：メールボックスに含まれている現在のメッセージ数。

- **Max Msgs**：メールボックスが保持することができるメッセージの最大数。これは構成時または作成時にセットされた値に一致します。
- **Msg Size**：プロセッサの最小アドレス可能データ単位（MADU）で表した各メッセージのサイズ。これは構成時または作成時にセットされた値に一致します。
- **Mem Segment**：メールボックスが配置されているメモリ・セグメントの名前。セグメント名を右クリックし、「Go To」コマンドを選択して「Kernel Object View」の該当 MEM セグメントを表示することができます。

3.5.3.5 セマフォ

セマフォ（SEM）の場合、次のようにプロパティが表示されます。

図 3-19. 「セマフォ」プロパティ

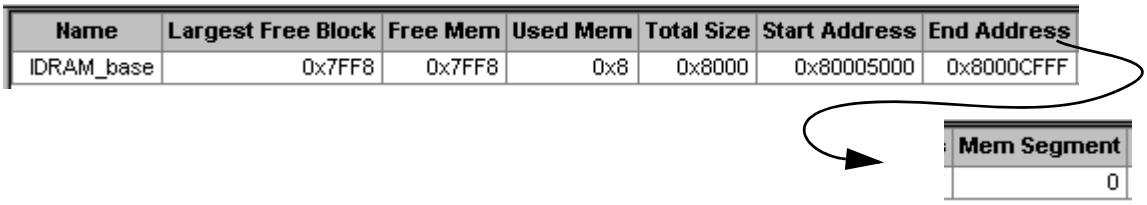
Name	Handle	Count	# Tasks Pending	Tasks Pending
0x8000e0dc	0x8000E0DC	0x2	0	
0x8000e004	0x8000E004	0x0	0	
0x8000dfdc	0x8000DFDC	0x4	0	

- **Name**：セマフォの名前。名前は、静的に構成されたセマフォの場合はラベルから取得され、動的に作成されたセマフォの場合は生成されます。ラベルは、構成時に指定した名前に一致します。
- **Handle**：ターゲット上のセマフォ・オブジェクト・ヘッダのアドレス。
- **Count**：現在のセマフォ・カウント。ブロックが行われる前に発生可能な保留されているセマフォの数を示します。
- **# Tasks Pending**：現在セマフォ上で保留されているタスクの数。
- **Tasks Pending**：セマフォ上で保留されているタスク名のプルダウン・リスト。選択したタスク上で右クリックし、「Go To」コマンドを選択して「Kernel Object View」の該当タスクを表示することができます。

3.5.3.6 メモリ

DSP/BIOS を使用すると、メモリ・セグメント・オブジェクトを構成することができます。セグメントには、メモリの動的割り当て元となるヒープが含まれる場合と含まれない場合があります。「Kernel Object View」は、メモリ・セグメント・オブジェクト内の動的メモリ・ヒープのプロパティ表示に重点を置いています。メモリ・セグメントおよびヒープ（MEM）の場合、次のようにプロパティが表示されます。

図 3-20. 「メモリ」プロパティ



Name	Largest Free Block	Free Mem	Used Mem	Total Size	Start Address	End Address
IDRAM_base	0x7FF8	0x7FF8	0x8	0x8000	0x80005000	0x8000CFFF

Mem Segment
0

- **Name**：構成されたメモリ・セグメント・オブジェクトの名前。
- **Largest Free Block**：このメモリ・セグメントのヒープ内で割り当てのために使用できる連続したメモリの最大量です。
- **Free Mem**：アプリケーションでは使用されず、ヒープから自由に割り当てることができるのメモリの合計量（MADU 単位）。
- **Used Mem**：すでにヒープから割り当て済みのメモリ量（MADU 単位）。この値が合計サイズになると、このフィールドが赤になり、警告が表示されます。
- **Total Size**：ヒープ内の最小アドレス可能単位（MADU）の合計数。
- **Start Address**：ヒープの先頭の位置。
- **End Address**：ヒープの最後の位置。
- **Mem Segment**：ヒープが配置されているメモリ・セグメントの数。

3.5.3.7 バッファ・プール

バッファ・プール（BUF）の場合、次のようにプロパティが表示されます。

図 3-21. 「バッファ・プール」プロパティ

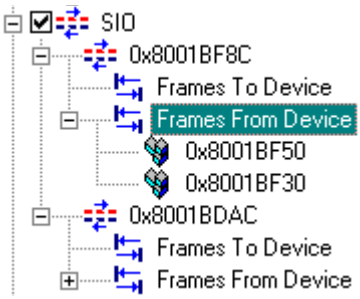
Name	Segment ID	Size of Buffer	# Buffers in Pool	# Free Buffers	Pool Start Address	Max Buffers Used
BUF0	ISRAM	8	2	2	0x254C	0

- **Name**：バッファ・プール・オブジェクトの名前。名前は、静的に構成されたタスクの場合はプールから取得され、動的に作成されたプールの場合は生成されます。ラベルは、構成時に指定した名前に一致します。
- **Segment ID**：バッファ・プールが存在するメモリ・セグメントの名前。
- **Size of Buffer**：バッファ・プール内部の各メモリ・バッファのサイズ（MADU 単位）。
- **# Buffers in Pool**：バッファ・プール内のバッファ数。

- **# Free Buffers :** バッファ・プール内で使用できる現在のバッファ数。
- **Pool Start Address :** バッファ・プールが開始するターゲット上のアドレス。
- **Max Buffers Used :** プール内で使用されているバッファ数。

3.5.3.8 ストリーム入出力（I/O）オブジェクト

「Kernel Object View」は、ストリーム入出力（I/O）（SIO）オブジェクトに関するさまざまなレベルの情報を提供します。SIO タイプの下のツリー内の最初のレベルは、構成された SIO オブジェクトまたはアプリケーション内で作成された SIO オブジェクトをリスト表示します。各 SIO オブジェクトの下には、「Frames To Device」および「Frames From Device」に対応した項目があり、さらに device->todevice および device->fromdevice キューに現在置かれているフレームのリストが含まれます。



ツリー内に表示される SIO オブジェクトの場合、次のようにプロパティが表示されます。

図 3-22. 「ストリーム入出力（I/O）」プロパティ

Name	Handle	Mode	I/O Model	Device...	Device ID	Num Frames to ...
inStreamSrc	0x80003D60	Input	Standard	scale	DTR_multiply	0x0
outStreamSrc	0x80003DD0	Output	Standard	pipe0		0x0

Num Frames from ...	Timeout	Buffer size	Number of buf...	Buffer Align..	Mem seg...
0x0	0xFFFFFFFF	0x80	0x3	0x1	0xFFFFFFFF
0x0	0xFFFFFFFF	0x80	0x3	0x1	0xFFFFFFFF

- **Name :** ストリーム入出力（I/O）オブジェクトの名前。名前は、静的に構成されたオブジェクトの場合はラベルから取得され、動的に作成されたオブジェクトの場合は生成されます。ラベルは、構成時に指定した名前に一致します。
- **Handle :** ターゲット上のストリーム入出力（I/O）オブジェクト・ヘッダのアドレス。
- **Mode**：Input（入力）または Output（出力）のいずれか。
- **I/O Model**：Standard（ストリーム作成時にバッファを割り当てる）または Issue/Reclaim（SIO_issue を使用してバッファの割り当てと提供を行う）。
- **Device Name**：接続デバイスの名前。DEV がイネーブルでなければ、空白になります。デバイス名を右クリックし、「Go To」コマンドを選択して「Kernel Object View」の該当デバイスを表示します。

- **Device ID**：接続デバイスの ID。
- **Num Frames to Device**：ToDevice キュー内のフレーム数。
- **Num Frames from Device**：FromDevice キュー内のフレーム数。
- **Timeout**：入出力（I/O）オペレーションのタイムアウト。
- **Buffer Size**：バッファのサイズ。
- **Number of Buffers**: ストリームのバッファ数。
- **Buffer Alignment**：I/O Model が Standard の場合は、メモリ・アライメント。
- **Mem Segment**：I/O Model が Standard の場合は、ストリーム・バッファを含むメモリ・セグメントの名前。

SIO オブジェクトの「Frames To Device」および「Frames From Device」またはそのいずれかのキューを選択すると、次のようにプロパティが表示されます。

図 3-23.　「SIO ハンドル」プロパティ

Name	Handle	Number of Elements
Frames To Device	0x800054F0	0x0
Frames From Device	0x800054F8	0x3

- **Name :** キューの名前。「Frames To Device」または 「Frames From Device」のいずれか。
- **Handle :** キューの最初のフレーム・エレメントのアドレス。
- **Number of Elements**: キュー内の現在のフレーム・エレメント数。

SIO キューの 1 つ以上のフレームを選択した場合は、次のようにプロパティが表示されます。

図 3-24.　「SIO フレーム」プロパティ

Name	Handle	Previous	Next
0x8001BF50	0x8001BF50	0x8001BF70	0x8001BF30
0x8001BF30	0x8001BF30	0x8001BF50	0x8001BF70

- **Name :** キューの名前。
- **Handle :** ターゲット上のキューのアドレス。
- **Previous**：キュー内の前のフレームのアドレス。
- **Next**：キュー内の次のフレームのアドレス。

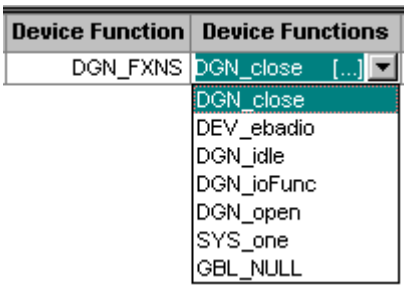
3.5.3.9 DEV デバイス

デバイス・オブジェクト（DEV）の場合、次のようにプロパティが表示されます。

図 3-25. 「デバイス」プロパティ

Name	Device ID	Type	Stream	Device Function	Device Functions	Device Parameters
printData	DGN_user	DEV_Fxns	0x0	DGN_FXNS	DGN_close [...]	0x80002950
sineWave	DGN_isine	DEV_Fxns	0x0	DGN_FXNS	DGN_close [...]	0x80002978
pipe0		DEV_Fxns	0x0	DPI_FXNS	text [...]	0x0
scale	DTR_multiply	DEV_Fxns	0x0	DTR_FXNS	0x6020 [...]	0x800007F4

- **Name**：デバイス・オブジェクトの名前。名前は、静的に構成されたデバイスの場合はプールから取得され、動的に作成されたデバイスの場合は生成されます。ラベルは、構成スクリプトで指定した名前に一致します。
- **Device ID**：デバイスの ID。
- **Type**：デバイスのタイプ。DEV_Fxns または IOM_Fxns のいずれかになります。
- **Stream**：デバイスが接続されている入出力（I/O）ストリーム。
- **Device Function**：デバイス関数のアドレス。
- **Device Function**：ここに表示されているリストのような関数テーブルに含まれている関数のプルダウン・リスト。
- **Device Parameters**：デバイス固有のパラメータのアドレス。



3.6 スレッドレベルのデバッグ方法

Code Composer Studio には、スレッドレベルでデバッグする機能が組み込まれています。この機能を使用すると、DSP/BIOS TSK や SWI をスレッド単位でデバッグすることができます。この機能を使用すると、デバッグするために選択したスレッドに対応する別の「Code Composer Studio」ウィンドウが開きます。このウィンドウでは、次のようなことができます。

- ❏ スレッドの実行と停止。
- ❏ スレッド内での実行を停止するためのブレークポイントのセットと解除。
- ❏ スレッドのコードのステップ実行。
- ❏ スレッドの現在の状態に関する情報の表示。
- ❏ メモリの読み出しと書き込み。

3.6.1 スレッドレベルのデバッグをイネーブルにする方法

DSP/BIOS は、スレッドレベルのデバッグを行うためのデフォルトのオペレーティング・システムです。他の DSP オペレーティング・システムとともに動作する場合にも、DSP/BIOS を再度選択する必要があります。Code Composer Studio は、次回のセッションで使用するために前回選択した OS の情報を保持しています。デフォルト OS を変更していた場合、DSP/BIOS に戻す手順は次のとおりです。

1) 「Tools」→「OS Selector」の順に選択して、「OS Selector」ウィンドウを開きます。

2) 「Current OS」ドロップダウン・リストから、「DSP/BIOS」を選択します。複数のバージョンの「DSP/BIOS」が表示されている場合には、ご使用になるバージョンを選択します。

3) 「OS Selector」ウィンドウで右クリックし、ポップアップ・メニューから「Close」を選択します。

スレッドレベルのデバッグを使用するには、各 Code Composer Studio のセッション時に最初に一度イネーブルにする必要があります。デバッグを行うアプリケーションをロードする前後で、次のようにして、この機能をイネーブルにすることができます。

1) 「Debug」→「Enable Thread-Level Debugging」の順に選択します （「Kernel Object View」が自動的に開き、スレッドレベルのデバッグがイネーブルになります）。

3.6.2 「Thread Control」ウィンドウを開く方法

スレッドレベルのデバックを開始する手順は次のとおりです。

1) Code Composer Studio 内でアプリケーションをロードし、実行します。

2) 「View」→「Threads」の順に選択します。アプリケーションから SWI および TSK のスレッドのリストを確認できます。

   ■ このリストは、ターゲットを停止すると常に更新されます。ただし、「Refresh Threads」を選択して、このリストを更新することができます。

- 10 を超えるスレッドがある場合、スレッドを選択するダイアログを開くことができます。
- 「Current Thread」を選択して、現在実行中のスレッドを選択します。
- このリストには、静的および動的に作成された TSK と SWI の両方が含まれています。
- このリストには、ソース・コードが提供されない DSP/BIOS カーネル・スレッドは含まれません。たとえば、KNL_swi、PRD_swi、および TSK_idle スレッドはデバッグすることができません。

3) スレッドを選択すると、別の「Code Composer Studio」ウィンドウが開きます。このウィンドウで、スレッドのデバッグを行います。

元の「Code Composer Studio」ウィンドウのタイトル・バーには、「CPU」と表示されます。このウィンドウは、「CPU Control」ウィンドウといい、アプリケーション全体をデバッグすることができます。

新しい「Code Composer Studio」ウィンドウでは、タイトル・バーに選択したスレッドの名前が表示されます。このウィンドウは、「Thread Control」ウィンドウといい、該当スレッドのみをデバッグすることができます。すべてのウィンドウでは、現在実行中のスレッドがステータス・バーに表示されます。「Thread Control」ウィンドウでは、ステータス・バーにウィンドウのスレッド状態が表示されます。

3.6.3 「Thread Control」ウィンドウを使用する方法

「CPU Control」ウィンドウでは、ターゲット・プロセッサ上で動作する 1 つのプログラムをデバッグすることができます。Code Composer Studio のデバッガには、ターゲット・プログラム内の DSP/BIOS スレッドがわかりません。ブレークポイントに到達すると、実行の継続が選択されるまで CPU は停止します。

「Thread Control」ウィンドウでは、Code Composer Studio の通常のデバッグ操作をすべて利用できます。ただし、実行されるオペレーションは該当スレッドのコンテキスト内で実行されるということに注意することが重要です。たとえば、Code Composer Studio デバッガは、「Thread Control」ウィンドウでデバッグするときに、ブレークポイントが現在のスレッドに適用できるかまず確認します。適用できない場合は、デバッガは自動的に次のブレークポイントまで実行を継続します。

スレッドレベルのデバッグに備わっている停止モードのため、ターゲット全体は CPU レベルで停止します。これにより、ターゲットが停止している間、DSP/BIOS は優先順位の高いスレッドを実行できません。このように、DSP/BIOS におけるスレッド・スケジュールは通常のリアルタイム動作から逸脱する場合があります。

「Thread Control」ウィンドウがアクティブになるのは、スレッドが存在する場合だけです。スレッドが終了した場合、ウィンドウ内のデバッグ・ツールはグレー表示されます。スレッドが再度ポストされるか、再度作成されると、ウィンドウは再びアクティブになります。

デバッグ動作は、「Thread Control」ウィンドウ内では次のように動作します。

- **Run と Step コマンド:** スレッドレベルのオペレーションの結果として、ターゲットが停止したとき、現在のスレッドの「CPU Control」ウィンドウと「Thread Control」ウィンドウだけが Run と Step コマンドを実行できます。

  DSP/BIOS API 関数をステップ実行できないことに注意してください。このような関数の場合、ソース・コードは提供されません。代わりに、デバッガはこれらの関数を自動的にステップ・オーバーします。DSP/BIOS 関数をステップイン要求したのにステップ・オーバーされてしまう場合、このような関数をステップ・オーバーすることを選択するよりも大幅に時間がかかります。

- **ブレークポイント：** 現時点でそのスレッドが実行中であるかどうかに関係なく、「CPU Control」ウィンドウと「Thread Control」ウィンドウのいずれでも、ブレークポイントのセットと解除を行うことができます。しかし、「Thread Control」ウィンドウ内でブレークポイントをセットしたり解除したりすると、該当スレッドのみが影響を受けます。別のスレッドが同じコードを実行していると、ブレークポイントは目に見えるようにトリガされません（ただし、ブレークポイントがセットされていないスレッドによって実行されている共有コード内でブレークポイントに到達すると、CPU は一時的に停止してから、再び自動的に実行します）。

- **Halt（停止）コマンド：**「CPU Control」ウィンドウと「Thread Control」ウィンドウのいずれでも、ターゲット実行時に停止コマンドを発行することができます。停止コマンドは、その発行場所に応じて、次のような結果になります。

スレッド・タイプ	実行中の場合	実行中ではない場合
SWI	現在実行している位置で停止。	（SWI が再開した時点ではなく）SWI が再入した時点で停止。「Thread Control」ウィンドウのステータス・バーには、このスレッドが対象になるまで「Thread Halting」が表示されます。
TSK	現在実行している位置で停止。	この TSK が実行を再開時に停止。「Thread Control」ウィンドウのステータス・バーには、このスレッドが対象になるまで「Thread Halting」が表示されます。

ターゲット・プログラムが停止すると、「Kernel Object View」のような停止モードのデバッグ・ツールなどを更新するためにデータはターゲットから読み出されます。

- **レジスタ操作：**ターゲットが停止すると、「CPU Control」ウィンドウと「Thread Control」ウィンドウは次のようにレジスタにアクセスできます。

スレッド・タイプ	実行中の場合	実行中ではない場合
SWI	すべてのレジスタへの完全な読み出しと書き込みアクセス。	レジスタにはアクセスしない。
TSK	すべてのレジスタへの完全な読み出しと書き込みアクセス。	コンテキストが保存されているレジスタにのみアクセス。

  利用できないレジスタ値には、0xBEEF とマークが付けらます。利用できないレジスタに書き込むとエラーになります。

- **メモリ操作：**メモリの読み出しと書き込みは、グローバルに「CPU Control」ウィンドウと「Thread Control」ウィンドウに適用されます。「Thread Control」ウィンドウでメモリ・ロケーションを変更すると、「CPU Control」ウィンドウで変更した場合と同じ影響を与えます。

- **コール・スタック表示：**タスクに対応した「Thread Control」ウィンドウは、コール・スタック内では TSK_exit を表示しません。この呼び出しは、「CPU Control」ウィンドウに表示されます。実行中の「Thread Control」ウィンドウは、「CPU Control」ウィンドウと同じコール・スタックを表示します。実行中ではない「Thread Control」ウィンドウは、コール・スタック上で実行を再開する関数を表示します。

3.7 フィールド・テスト用の計測

組み込みの DSP/BIOS ランタイム・ライブラリおよび DSP/BIOS 解析ツールは、実動システムで実行中のプログラムと相互作用する新世代のテストおよび診断ツールをサポートしています。DSP/BIOS による計測は非常に効率が高いので、実動プログラムでは、製造テストおよびフィールド診断ツールに使用する明示計測を保持しておくことができます。これは、暗黙計測および明示計測の両方と相互作用するように設計できます。

3.8 リアルタイム・データ・エクスチェンジ

リアルタイム・データ・エクスチェンジ（RTDX）を使用すると、DSP アプリケーションが現実世界でどのように働くかを、リアルタイムで継続的に見ることができます。RTDX プラグインを使用することにより、システム開発担当者は、ターゲット・アプリケーションを妨害せずに、ホスト・コンピュータと DSP デバイスの間でデータを転送することができます。データは、ホストで任意の OLE オートメーション・クライアントを使用して解析し、ビジュアル化できます。これによりシステムが実際にどのように動作するかを現実に即した方法で理解できるので、開発時間を短縮することができます。

注 :

RTDX は、新しい DSP デバイスまたはボードの初期のリリースではサポートされていないことがあります。

RTDX は、ターゲット・コンポーネントとホスト・コンポーネントの両方から構成されています。ターゲット DSP では、小さい RTDX ソフトウェア・ライブラリが実行されます。DSP アプリケーションは、このライブラリの API に対する関数呼び出しを使用して、このライブラリとの間で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、JTAG インターフェイスを介してホスト・プラットフォームとの間でデータをやり取りします。ホストへのデータ転送は、DSP アプリケーションの実行中にリアルタイムで行われます。

ホスト・プラットフォームでは、RTDX ホスト・ライブラリは Code Composer Studio と共同して作業を行います。画面および解析ツールと RTDX の間では使いやすい COM API を介して、ターゲット・データを入手するための、および DSP アプリケーションにデータを送信するための通信が行われます。設計者は、次に示す標準ソフトウェア表示パッケージを任意に選択して使用できます。

- ❏ National Instruments の LabVIEW
- ❏ Quinn-Curtis の Real-Time Graphics Tools
- ❏ Microsoft Excel

また、独自の Visual Basic または Visual C++ アプリケーションを開発することもできます。データの取得に重点を置く代わりに、データを最もわかりやすい形でビジュアル化できるように画面を設計することに重点を置くことができます。

3.8.1 RTDX アプリケーション

RTDX は、多種多様な制御、サーボ、およびオーディオ・アプリケーションに適しています。たとえば無線通信のメーカでは、ボコーダ・アルゴリズムの出力をキャプチャしてスピーチ・アプリケーションの実装を検査することができます。

RTDX は、組み込み制御システムにも利点を提供します。ハードディスク・ドライブの設計者は、サーボ・モータへの不適正な信号によってドライブをクラッシュさせることなく、アプリケーションをテストできます。エンジン制御の設計者は、制御アプリケーションの実行中に、変化する要因(熱や環境条件など)を解析できます。

これらのどのアプリケーションの場合も、情報を最もわかりやすい形で表示するビジュアル化ツールを選択することができます。

3.8.2 RTDX の使用

RTDX は、DSP/BIOS の中でも、また DSP/BIOS なしでも使用できます。c:\ti\tutorial フォルダ・ツリーの中の volume4、hostio1、および hostio2 の例に含まれているプログラムでは、RTDX を各種 DSP/BIOS モジュールと一緒に使用しています。*c:\ti*\examples*target*\rtdx フォルダ・ツリーの中の例では、DSP/BIOS を使わずに RTDX を使用しています。

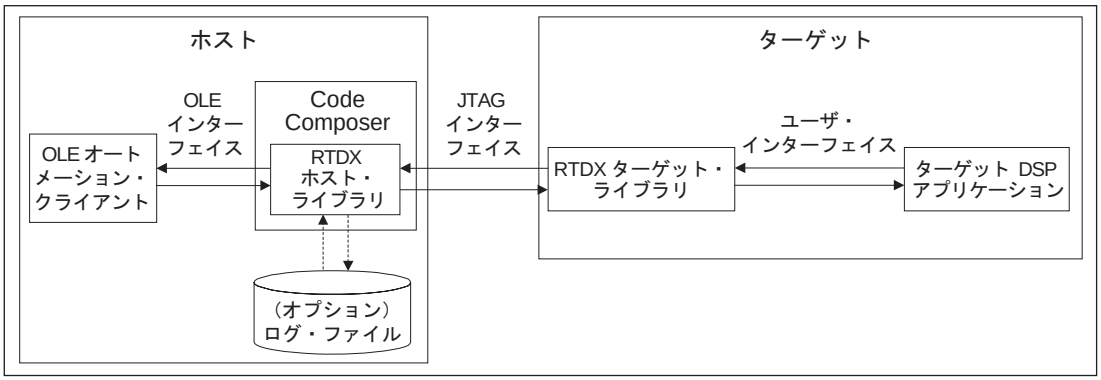
RTDX は、Windows 98 または Windows NT バージョン 4.0 が稼働している PC をホストとする Code Composer Studio で使用できます。RTDX はシミュレーションの中でも使用できます。

本書では、読者が C、Visual Basic または Visual C++、および OLE/ActiveX のプログラミングに精通していることを前提としています。

3.8.3 RTDX データ・フロー

ホスト（PC）とターゲット（TI プロセッサ）間の Code Composer Studio のデータ・フローを図 3-26 に示します。

図 3-26.　ホストとターゲット間の RTDX データ・フロー



3.8.3.1 ターゲットからホストへのデータ・フロー

ターゲットにデータを記録するには、出力チャネルを宣言し、ユーザ・インターフェイスの中で定義されているルーチンを使用して、そこにデータを書き込む必要があります。このデータは、RTDX ターゲット・ライブラリ内に定義されている RTDX ターゲット・バッファに即時に記録されます。そして、そのバッファ内のデータは JTAG インターフェイスを介してホストに送られます。

RTDX ホスト・ライブラリは、JTAG インターフェイスからこのデータを受信し、記録します。ホストは、データをメモリ・バッファまたは RTDX ログ・ファイルに記録します（これは、指定されている RTDX ホスト記録モードによって決まります）。

記録されたデータは、OLE オートメーション・クライアントとなっている任意のホスト・アプリケーションにより検索することができます。OLE 対応ホスト・アプリケーションの代表的な例を次に示します。

- ❏ Visual Basic アプリケーション
- ❏ Visual C++ アプリケーション
- ❏ Lab View
- ❏ Microsoft Excel

一般に、RTDX OLE オートメーション・クライアントは、データをわかりやすい方法でビジュアル化するためのディスプレイです。

3.8.3.2 ホストからターゲットへのデータ・フロー

ターゲットがホストからデータを受信するには、まず入力チャネルを定義し、ユーザ・インターフェイスの中で定義されているルーチンを使用してそこからデータを要求する必要があります。データに対する要求は RTDX ターゲット・バッファに記録され、JTAG インターフェイスを介してホストに送信されます。

OLE オートメーション・クライアントは、OLE インターフェイスを使用してターゲットにデータを送信することができます。ターゲットに送信されたデータは、すべて RTDX ホスト・ライブラリ内のメモリ・バッファに書き込まれます。RTDX ホスト・ライブラリがターゲット・アプリケーションから読み取り要求を受け取ると、JTAG インターフェイスを介してホスト・バッファ内のデータがターゲットに送信されます。このデータは、ターゲット上の要求された位置にリアルタイムで書き込まれます。オペレーションが完了すると、ホストは RTDX ターゲット・ライブラリに通知します。

3.8.3.3 RTDX ターゲット・ライブラリ・ユーザ・インターフェイス

ユーザ・インターフェイスは、ターゲット・アプリケーションと RTDX ホスト・ライブラリ間でデータを交換するための最も安全な方法を提供します。

ユーザ・インターフェイスの中で定義されているデータ型と関数は、次のことを行います。

- ターゲット・アプリケーションが、RTDX ホスト・ライブラリにデータを送信できるようにします。
- ターゲット・アプリケーションが、RTDX ホスト・ライブラリからのデータを要求できるようにします。
- ターゲット上でのデータ・バッファを提供します。データのコピーが、ホストに送信される前にターゲット・バッファに保存されます。この処置によりデータの保全性が確保され、リアルタイムの干渉が最小限に抑制されます。
- 割り込みの安全性を提供します。ユーザ・インターフェイスの中で定義されているルーチンを、割り込みハンドラの中で呼び出すことができます。
- 通信メカニズムが正しく使用されるようにします。ホストとターゲットの間で、JTAG インターフェイスを使用して一度に 1 つだけデータを交換できるようにすることが、必須条件です。ユーザ・インターフェイスの中で定義されているルーチンは、下位レベルのインターフェイスの呼び出しタイミングを操作します。

3.8.3.4 RTDX ホスト OLE インターフェイス

OLE インターフェイスは、OLE オートメーション・クライアントが RTDX ホスト・ライブラリと通信できるようにするための方式を記述します。

OLE インターフェイス内で定義されている関数は、次のことを行います。

- OLE オートメーション・クライアントが、RTDX ログ・ファイルに記録されたデータ、または RTDX ホスト・ライブラリによりバッファリングされているデータにアクセスできるようにします。
- OLE オートメーション・クライアントが、RTDX ホスト・ライブラリを介してデータをターゲットに送信できるようにします。

3.8.4 RTDX モード

RTDX ホスト・ライブラリでは、次のいずれかのモードでターゲット・アプリケーションからデータを受信できます。

- ❑ **不連続**： データはホスト上のログ・ファイルに書き込まれます。不連続モードは、有限量のデータをキャプチャして、ログ・ファイルに記録したい場合に使用します。
- ❑ **連続**: データは RTDX ホスト・ライブラリによりバッファに入れられるだけです。ログ・ファイルには書き込まれません。連続モードは、DSP アプリケーションからのデータを連続的に取得および表示したい場合であって、しかもデータをログ・ファイルに保存する必要がない場合に使用します。

注 :

バッファの内容を排出してターゲットからデータが連続的に流れるようにするために、OLE オートメーション・クライアントは、個々のターゲット出力チャネルから連続的にデータを読み込む必要があります。この制約条件を満たすことができない場合は、ターゲットからのデータ・フローが途切れてしまい、結果的にデータ速度が低下し、場合によってはチャネルがデータを取得できなくなることがあります。さらに OLE オートメーション・クライアントは、チャネルでのデータの損失を防ぐために、スタートアップ時にすべてのターゲット出力チャネルをオープンする必要があります。

3.8.5 アセンブリ・コードを記述するときの特別な注意事項

ユーザ・ライブラリ・インターフェイス内の RTDX 機能には、アセンブリ・コードで記述されたターゲット・アプリケーションによりアクセスできます。

ご使用のプラットフォームに該当する C 言語の呼び出し規則、ランタイム環境、およびランタイム・サポート関数については、『TMS320C54x オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル』、『TMS320C55x オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル』、または『TMS320C6000 オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル』を参照してください。

3.8.6 ターゲット・バッファのサイズ

RTDX ターゲット・バッファは、ホストに転送されるのを待っているデータを一時的に保存するために使用されます。少量のデータのみを転送する場合は、バッファのサイズを小さくすることができます。逆に、デフォルトのバッファ・サイズより大きいデータ・ブロックを転送する場合は、バッファのサイズを大きくする必要があります。

Tconf スクリプトの RTDX バッファ・サイズを変更することができます。

3.8.7 ターゲットからホストへまたはホストからターゲットへのデータ送信

ユーザ・ライブラリ・インターフェイスは、次のことを行うためのデータ型と関数を提供します。

- ターゲットからホストへのデータの送信
- ホストからターゲットへのデータの送信

次のデータ型および関数は、ヘッダ・ファイル rtdx.h の中で定義されています。これらは、DSP/BIOS を介して、またはスタンドアロンで使用することができます。

- 宣言マクロ
  - RTDX_CreateInputChannel
  - RTDX_CreateOutputChannel
- 関数
  - RTDX_channelBusy
  - RTDX_disableInput
  - RTDX_disableOutput
  - RTDX_enableOutput
  - RTDX_enableInput
  - RTDX_read
  - RTDX_readNB
  - RTDX_sizeofInput
  - RTDX_write
- マクロ
  - RTDX_isInputEnabled
  - RTDX_isOutputEnabled

すべての RTDX 関数の詳しい説明は、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』を参照してください。

第 4 章

スレッド・スケジューリング

本章では、DSP/BIOS プログラムで使用できるスレッドの型、スレッドの動作、およびプログラム実行時のスレッドの優先順位について解説します。

項目 **ページ**

4.1 スレッド・スケジューリングの概要

多くのリアルタイム DSP アプリケーションは、多くの場合データの可用性や制御信号の存在などの外部イベントに対する応答として、互いに無関係のように見える多数の関数を同時に実行する必要があります。実行する関数自体と、それをどの時点で実行するかは、どちらも重要な要素です。

このような関数をスレッドと呼びます。スレッドは、システムによって広義に定義されている場合と狭義に定義されている場合があります。DSP/BIOS ではこの用語は広義に定義されていて、DSP が実行する命令で構成される任意の独立したストリームを意味します。スレッドは、サブルーチン、割り込みサービス・ルーチン（ISR）、または関数呼び出しを内部に組み込むこともできる単一管理点です。

DSP/BIOS を使用すると、アプリケーションをスレッドの集合体として構築できます。個々のスレッドはモジュール化された 1 つの機能を実行します。マルチスレッド・プログラムを単一のプロセッサで実行するために、優先順位の高いスレッドを優先順位の低いスレッドより優先的に実行したり、ブロッキング、通信、同期など、スレッド間でさまざまなタイプの相互作用ができるようになっています。

このようにモジュラ方式で構成されたリアルタイム・アプリケーション・プログラムは、たとえば単一の集中方式のポーリング・ループに比べて、設計、実装、保守が簡単になります。

DSP/BIOS は、優先順位の異なるいくつかの型のプログラム・スレッドをサポートします。スレッドの型ごとに、実行特性および優先実行特性が異なります。スレッドの型には次のものがあります（優先順位の高い方から順に列挙してあります）。

- **ハードウェア割り込み (HWI)**：CLK 関数が含まれます。
- **ソフトウェア割り込み (SWI)**：PRD 関数が含まれます。
- **タスク（TSK）**
- **バックグラウンド・スレッド（IDL）**

次の節では、これらのスレッドの型について簡単に説明し、その後の各節でそれぞれについてさらに詳しく説明します。

4.1.1 スレッドの型

DSP/BIOS プログラムで使用するスレッドには、次の 4 つの主な型があります。

- **ハードウェア割り込み（HWI）**：DSP 環境内で発生する外部非同期イベントに対する応答してトリガされます。HWI 関数（割り込みサービス・ルーチンまたは ISR とも呼ばれます）は、ハードウェア割り込みがトリガされた後で、デッドラインを厳守しなければならない重要なタスクを処理するために実行されます。HWI 関数は DSP/BIOS アプリケーションの中で最も優先順位の高いスレッドです。200 MHz で動作する DSP の場合、HWI は、200 kHz に近い周波数で実行し、かつ 2 ～ 100 マイクロ秒というデッドライン内に完了することが必要なアプリケーション・タスクに使用します。もっと高速な DSP の場合、HWI は比較的高い周波数で実行し、かつデッドラインが比較的に短いタスクに使用します。ハードウェア割り込みの詳細については、4.2 節「ハードウェア割り込み」（4-11 ページ）を参照してください。

- **ソフトウェア割り込み（SWI）**： ハードウェア割り込み（HWI）に似ています。HWI はハードウェア割り込みによりトリガされますが、ソフトウェア割り込みはプログラムが SWI 関数を呼び出すことによりトリガされます。ソフトウェア割り込みは、ハードウェア割り込みと TSK の間の追加の優先順位レベルを提供します。SWI が取り扱うのは、時間的制約があるためタスクとして実行することはできないが、デッドラインはハードウェア ISR ほど厳しくないというスレッドです。HWI と同様に、SWI のスレッドも完了するまで実行されます。ソフトウェア割り込みは、デッドラインが 100 マイクロ秒以上のイベントをスケジュールするために使用します。SWI を使用すると、HWI は重要度の低い処理を優先順位の低いスレッドに移せるため、CPU が割り込みサービス・ルーチン内部での処理に要する時間を最小限に抑えることができ、したがって他の HWI がディセーブルにされる可能性を低くすることができます。ソフトウェア割り込みの詳細については、4.3 節「ソフトウェア割り込み」（4-26 ページ）を参照してください。

- **タスク（TSK）:** バックグラウンド・スレッドより優先順位が高く、ソフトウェア割り込みより優先順位が低いスレッドです。タスクとソフトウェア割り込みとの違いは、タスクは必要なリソースが使用可能になるまで実行を一時中断できることです。DSP/BIOS は、タスク間の通信および同期に使用できる構造体を提供します。このような構造体には、キュー、セマフォ、メールボックスなどがあります。タスクの詳細については、4.4 節「タスク」（4-39 ページ）を参照してください。

- **バックグラウンド・スレッド**: DSP/BIOS アプリケーション内で、最も低い優先順位でアイドル・ループ（IDL）を実行します。DSP/BIOS アプリケーションは、main に復帰後に DSP/BIOS の各モジュールのスタートアップ・ルーチンを呼び出し、次にアイドル・ループに入ります。アイドル・ループは、IDL オブジェクト用のすべての関数を呼び出す連続ループです。ループ内の各関数は、他のすべての関数の実行が終了するまでは再度呼び出されません。アイドル・ループは、それより優先順位の高いスレッドにより優先権を奪われない限り、連続的に実行されます。アイドル・ループの中では、条件の厳しいデッドラインのない関数のみを実行するようにします。バックグラウンド・タスクの詳細については、4.5 節「アイドル・ループ」（4-49 ページ）を参照してください。

DSP/BIOS プログラムの中で使用できる関数には、他にもいくつかの種類があります。これらの関数は、上記に挙げたスレッドの型のいずれかのコンテキストの中で実行されます。

- **クロック（CLK）関数**： オンデバイス・タイマの割り込みの頻度でトリガされます。デフォルトでは、これらの関数はハードウェア割り込みによりトリガされ、HWI 関数として実行されます。詳細については、4.9 節「タイマ、割り込み、およびシステム・クロック」（4-71 ページ）を参照してください。

- **周期（PRD）関数**: オンデバイス・タイマ割り込み、またはその他の何らかのイベントの発生の倍数に基づいて実行されます。周期関数は、特殊タイプのソフトウェア割り込みです。詳細については、4.10 節「周期関数マネージャ（PRD）とシステム・クロック」（4-77 ページ）を参照してください。

- **データ通知関数**： ユーザがパイプ（PIP）またはホスト・チャネル（HST）を使用してデータを転送するときに実行されます。これらの関数は、データ・フレームがリードまたはライトされるときに、ライタまたはリーダに通知するためにトリガされます。データ通知関数は、PIP_alloc、PIP_get、PIP_free、または PIP_put を呼び出した関数のコンテキストの一部として実行されます。

4.1.2 使用するスレッドの型を選択する方法

アプリケーション・プログラム内の各スレッドについて、どのような型と優先順位レベルを選択するかによって、スレッドが所定の時刻にスケジュールされ正しく実行されるかどうかが決まります。DSP/BIOS は静的に構成されているので、スレッドの型を別の型に簡単に変更することができます。

プログラムが実行する各タスクに使用するオブジェクトの型を決定するには、次のような規則があります。

- **SWI または TSK と HWI**：ハードウェア割り込みサービス・ルーチンの中の重要な処理のみを行います。HWI は、デッドラインを守れないとデータが書き換えられるような、デッドラインが 5 マイクロ秒以下のハードウェア割り込み（IRQ）を処理する場合に使用します。ソフトウェア割り込みまたはタスクは、デッドラインが 100 マイクロ秒程度以上のイベントに使用します。優先順位の低い処理を行うためには、HWI 関数でソフトウェア割り込みまたはタスクをポストする必要があります。優先順位の低いスレッドを使用すると、割り込みがディセーブルにされる時間（割り込みレイテンシ）が最小限に抑えられるので、他のハードウェア割り込みを実行できる可能性が高まります。

- **SWI と TSK :** 相互依存関係とデータ共有の要件が比較的単純な関数の場合は、ソフトウェア割り込みを使用するようにします。要件が複雑な場合は、タスクを使用します。優先順位の高いスレッドは優先順位の低いスレッドの優先権を奪うことができますが、他のイベントを（たとえばリソースが使用可能になるまで）待つために一時中断できるのはタスクだけです。SWI よりタスクの方が、共有データを使用するためのオプションを多くもっています。ソフトウェア割り込みの関数で必要なすべての入力は、プログラムが SWI をポストする時点で準備ができていなければなりません。SWI オブジェクトのメールボックス構造体は、リソースが使用可能になるタイミングを判別する手段を提供します。SWI はすべて 1 つのスタックから実行されるので、メモリ使用効率が高くなります。

- **IDL**：バックグラウンド関数は、他の処理が必要でないときに重要度の低いハウスキーピング・タスクを実行するために作成します。IDL 関数には、厳密なデッドラインがないのが普通です。代わりに、この関数はシステムのプロセッサに空き時間があるときには必ず実行されます。

- **CLK**： CLK 関数は、タイマ割り込みにより関数を直接トリガしたい場合に使用します。これらの関数は HWI 関数として実行されるので、最小限の処理時間で完了するようにしなければなりません。デフォルトの CLK オブジェクトである PRD_clock では、周期関数用として 1 ティックが設定されます。同じ速度で実行する追加の CLK オブジェクトを加えることもできます。しかし、CLK 関数は HWI 関数として実行されるので、すべての CLK 関数を実行するために必要な時間を最小限に抑える必要があります。

- **PRD**： この関数は、オンデバイス・タイマの低分解能速度または他のイベント（外部割り込みなど）に基づく速度の倍数で関数を実行したい場合に使用します。これらの関数は、SWI 関数として実行されます。

- **PRD と SWI**：すべての PRD 関数は同じ SWI 優先順位で実行されるので、ある PRD 関数が他の PRD 関数の優先権を奪うことはできません。ただし、PRD 関数は、処理時間の長いルーチンに対して優先順位の低いソフトウェア割り込みをポストできます。これにより、次のシステム・ティックが発生して PRD_swi が再度ポストされたときに、PRD_swi ソフトウェア割り込みは処理時間の長いルーチンから優先権を奪うことができます。

4.1.3 スレッド特性の比較

表 4-1 に、DSP/BIOS がサポートするスレッドの型の比較を示します。

表 4-1. スレッド特性の比較

特性	HWI	SWI	TSK	IDL
優先順位	最高	2 番目	3 番目	最低
優先順位レベルの数	DSP に依存	15。周期関数は PRD_swi SWI オブジェクトの優先順位に従って実行されます。タスク・スケジューラは最も低い優先順位で実行されます。	16（ID ループ用の 1 個を含む）	1
譲渡および保留の可否	不可。他の優先実行権が発生した場合以外は、完了まで実行されます。	不可。他の優先実行権が発生した場合以外は、完了まで実行されます。	可	望ましくない。PC がターゲット情報を取得できなくなります。
実行状態	インアクティブ、レディ、実行中	インアクティブ、レディ、実行中	レディ、実行中、ブロック状態、終了	レディ、実行中
スケジューラをディセーブルにする要因	HWI_disable	SWI_disable	TSK_disable	プログラム出口
ポストするか実行可能にする要因	割り込みの発生	SWI_post、SWI_andn、SWI_dec、SWI_inc、SWI_or	TSK_create	main() 出口（ただし現在実行中の他のスレッドがない場合）
使用されるスタック	システム・スタック（1 プログラムにつき 1 個）	システム・スタック（1 プログラムにつき 1 個）	タスク・スタック（1 プログラムにつき 1 個）	デフォルトではタスク・スタックを使用（注 1 を参照）

注： 1) TSK マネージャをディセーブルにした場合、IDL スレッドはシステム・スタックを使用します。

表 4-1.　スレッド特性の比較（続き）

特性	HWI	SWI	TSK	IDL
他のスレッドの優先権を奪うときにセーブされるコンテキスト	カスタマイズ可能	特定のレジスタはシステム・スタックにセーブされる（注 2 を参照）	コンテキスト全体がタスク・スタックにセーブされる	-- 適用外 --
ブロックされたときにセーブされるコンテキスト	-- 適用外 --	-- 適用外 --	C レジスタ・セットがセーブされる（ご使用のプラットフォームに対応した『オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル』を参照）	-- 適用外 --
スレッドとのデータ共有を仲介するもの	ストリーム、キュー、パイプ、グローバル変数	ストリーム、キュー、パイプ、グローバル変数	ストリーム、キュー、パイプ、ロック、メールボックス、グローバル変数	ストリーム、キュー、パイプ、グローバル変数
スレッドとの同期を仲介するもの	-- 適用外 --	SWI メールボックス	セマフォ、メールボックス	-- 適用外 --
関数フック	なし	なし	あり：initialize、create、delete、exit、task switch、ready	なし
静的作成	デフォルトの構成テンプレートに組み込み	可	可	可
動的作成	可（注 3 を参照）	可	可	なし
優先順位の動的変更	不可（注 4 を参照）	可	可	なし
暗黙ロギング	なし	ポストおよび完了イベント	レディ、起動、ブロック、再開、終了イベント	なし
暗黙統計	監視対象値	実行時間	実行時間	なし

2) セーブされるレジスタのリストについては、4.3.7　項「ソフトウェア割り込みの優先権奪取時にレジスタをセーブする方法」（4-36 ページ）を参照してください。

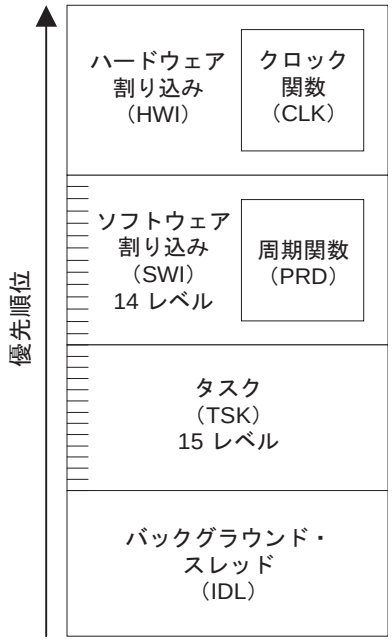
3) HWI オブジェクトは DSP 割り込みに対応しているので、動的に作成することはできません。ただし、割り込み関数は実行時に変更することができます。

4) HWI 関数で HWI_enter を呼び出す場合は、HWI 関数の実行中にどの割り込みをイネーブルにするかを示すビットマスクをこの関数で渡すことができます。イネーブルされた割り込みは、現行の割り込みよりそのイネーブルされた割り込みの優先順位が低かった場合でも、HWI 関数を優先実行することができます。

4.1.4 スレッドの優先順位

DSP/BIOS の中で優先順位が最も高いのはハードウェア割り込みです。HWI オブジェクトの中の優先順位は、DSP/BIOS で暗黙に維持されるわけではありません。この HWI の優先順位は、特定の CPU サイクルの中でレディ状態になった複数の割り込みに対して CPU がサービスを提供する場合に、その処理順序に適用されるだけです。ハードウェア割り込みの場合は、CSR の GEI ビットをリセットするかまたはこれに対応する IER のビットをセットすることによって別の割り込みをディセーブルにしない限り、別の割り込みが優先実行されます。

図 4-1. スレッドの優先順位



ソフトウェア割り込みは、ハードウェア割り込みより優先順位が低くなります。ソフトウェア割り込みには、14 個の優先順位レベルを使用することができます。ソフトウェア割り込みは、それより優先順位の高いソフトウェア割り込みまたはすべてのハードウェア割り込みにより優先権を奪われます。ソフトウェア割り込みはブロックできません。

タスクは、ソフトウェア割り込みより優先順位が低くなります。タスクには、15 個の優先順位レベルがあります。タスクは、それより優先順位の高いスレッドにより優先権を奪われることがあります。タスクはブロック可能で、リソースの可用性、または自身より優先順位の低いスレッドが完了するまで待機することもできます。

バックグラウンド・アイドル・ループは最も優先順位の低いスレッドで、CPU が他のスレッドに使用されていないときにループ内で実行されます。CPU が他のスレッドに使用されていないときに、ループ内で実行されます。

4.1.5 優先権の譲渡と優先実行

DSP/BIOS のスケジューラは、次の場合を除き、実行可能な状態にある最も優先順位の高いスレッドを実行します。

- ❑ 実行中のスレッドが、一時的に一部またはすべての割り込みをディセーブルにしているため（HWI_disable または HWI_enter を使用）、ハードウェア ISR が実行できない状態になっている場合
- ❑ 実行中のスレッドが、一時的にソフトウェア割り込みをディセーブルにしている場合（SWI_disable を使用）。この場合、優先順位の高いソフトウェア割り込みでも現在のスレッドの優先権を奪うことはできません。ただし、ハードウェア割り込みが現在のスレッドから優先権を奪うのを阻止することはありません。
- ❑ 実行中のスレッドが、一時的にタスク・スケジューリングをディセーブルにしている場合（TSK_disable を使用）。この場合、優先順位の高いタスクでも現在のタスクの優先権を奪うことはできません。ただし、ソフトウェア割り込みまたはハードウェア割り込みが現在のスレッドの優先権を奪うのを阻止することはありません。
- ❑ 最も優先順位の高いスレッドがブロックされているタスクである場合。この状況が生じるのは、タスクが TSK_sleep、LCK_pend、MBX_pend、または SEM_pend を呼び出した場合です。

ハードウェア割り込みとソフトウェア割り込みは両方とも、DSP/BIOS タスク・スケジューラと対話できます。タスクがブロックされるときは、多くの場合セマフォが使用不能でタスクが保留状態になっていることが原因です。セマフォは他のタスクからだけでなく、HWI や SWI からもポストされます。HWI または SWI が保留されているタスクのブロックを解除するためにセマフォをポストした場合、そのタスクの方が現在実行中のタスクより優先順位が高ければ、プロセッサはそのタスクに処理を切り換えます。

HWI または SWI を実行しているときには、DSP/BIOS は「システム・スタック」と呼ばれる専用のシステム割り込みスタックを使用します。各タスクは、それぞれ専用スタックを使用します。したがって、システム内に TSK タスクがない場合は、すべてのスレッドが同じシステム・スタックを共有します。DSP/BIOS は個々のタスクごとに専用のスタックを使用するので、アプリケーションとタスク・スタックの両方をさらに小さくすることができます。システム・スタックの方が小さいので、貴重な高速メモリにこれを配置することができます。

表 4-2 に、ある型のスレッド（最上部の行）が実行中に別のスレッド（左端の列）が実行可能状態になったときに起きる内容を示します。結果は、実行可能状態になった型のスレッドがイネーブルにされているかディセーブルにされているかによって異なります （示されている動作は、実行可能状態になったスレッドのものです）。

表 4-2.　スレッドの優先順位

ポストされたスレッド	実行中のスレッド HWI	SWI	TSK	IDL
イネーブルにされている HWI	優先実行	優先実行	優先実行	優先実行
ディセーブルにされている HWI	再イネーブルまで待機	再イネーブルまで待機	再イネーブルまで待機	再イネーブルまで待機
イネーブルにされた優先順位の高い SWI	——	優先実行	優先実行	優先実行
ディセーブルにされている SWI	待機	再イネーブルまで待機	再イネーブルまで待機	再イネーブルまで待機
優先順位の低い SWI	待機	待機	——	——
イネーブルにされた優先順位の高い TSK	——	——	優先実行	優先実行
ディセーブルにされている TSK	待機	待機	再イネーブルまで待機	再イネーブルまで待機
優先順位の低い TSK	待機	待機	待機	——

図 4-2 に、SWI および HWI がイネーブルにされていて（デフォルト）、ハードウェア割り込みルーチンが、現在実行中のソフトウェア割り込みより優先順位の高いソフトウェア割り込みをポストした場合の実行グラフを示します。また、最初の ISR 実行中に、第 2 のハードウェア割り込みも生じています。最初の ISR が実行中は第 2 の割り込みをマスク・オフしてしまう（つまりディセーブルにする）ので、第 2 の ISR は保留されます。

図 4-2.　優先実行のシナリオ

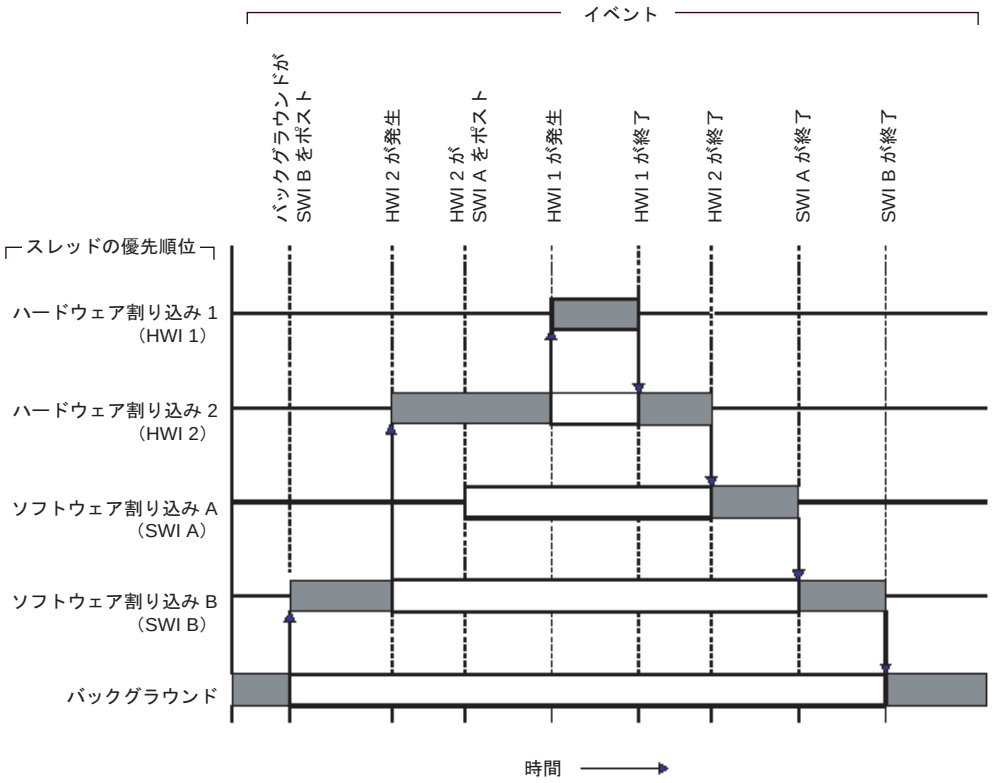


図 4-2 に示すように、優先順位の低いソフトウェア割り込みは、ハードウェア割り込みにより非同期に優先権を奪われます。最初の ISR が優先順位の高いソフトウェア割り込みをポストしていますが、このソフトウェア割り込みは、両方のハードウェア割り込みルーチンの実行が完了してから実行されます。

4.2 ハードウェア割り込み

ハードウェア割り込みは、アプリケーションが外部非同期イベントに応答して実行しなければならない重要な処理を取り扱います。ハードウェア割り込みの管理には、DSP/BIOS HWI モジュールが使用されます。

代表的な DSP システムでは、ハードウェア割り込みは、オンデバイス・ペリフェラル・デバイスか、または DSP の外部にあるデバイスによってトリガされます。どちらの場合も、割り込みによりプロセッサは ISR アドレスに方向を変えます。DSP/BIOS HWI オブジェクトにより生じる割り込みの方向変更先のアドレスは、ユーザ・ルーチンまたは共通システム HWI ディスパッチャの場合もあります。

ハードウェア ISR は、アセンブリ言語、C、またはその両方を組み合わせて記述することができます。HWI 関数は、効率性を考慮してアセンブリ言語で記述するのが一般的です。HWI オブジェクトの関数を完全に C のみで記述するには、システム HWI ディスパッチャを使用する必要があります。

すべてのハードウェア割り込みは、完了するまで実行されます。ISR に実行の機会が与えられる前に HWI が複数回ポストされたとしても、ISR は 1 回しか実行されません。そのため、1 つの HWI 関数が実行するコードの量を最小限に抑える必要があります。GIE ビットがイネーブルにされている場合、ハードウェア割り込みは、IEMASK によりイネーブルにされている割り込みにより優先実行権を奪われることがあります。

HWI 関数が PIP API のいずれか（PIP_alloc、PIP_free、PIP_get、PIP_put）を呼び出した場合は、HWI コンテキストの一環としてパイプの notifyWriter 関数または notifyReader 関数が実行されます。

注 :

HWI オブジェクトを C 関数と合わせて使用する場合は、*interrupt* キーワードまたは INTERRUPT プラグマを使用しては**いけません**。HWI_enter/HWI_exit マクロおよび HWI ディスパッチャにはこの機能が含まれているので、C 修飾子を使用すると破滅的な結果を招く可能性があります。

4.2.1 割り込みを構成する方法

HWI マネージャは、DSP/BIOS 構成テンプレートに、DSP 内の各ハードウェア割り込みにつき HWI オブジェクトを 1 つずつ含めます。

DSP 内の各ハードウェア割り込みごとに、ISR を構成できます。ユーザがしなければならないことは、Tconf スクリプトで、該当の HWI オブジェクトのハードウェア割り込みに応答して呼び出される ISR の名前を入力することだけです。各ハードウェア割り込みがそれぞれ適切な ISR により処理されるように割り込みテーブルを設定するのは、DSP/BIOS の役目です。また、割り込みテーブルを配置するメモリ・セグメントを構成することもできます。

HWI オブジェクトとそのパラメータに関する説明は、DSP/BIOS オンライン・ヘルプに含まれています。HWI モジュールの API コールの詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「HWI Module」を参照してください。

4.2.2 ハードウェア割り込みをディセーブルまたはイネーブルにする方法

ソフトウェア割り込みまたはタスクの中で、重要な処理セクションを実行しているときに、ハードウェア割り込みを一時的にディセーブルにすることができます。割り込みをディセーブルにしたりイネーブルにしたりするには、HWI_disable 関数と HWI_enable/HWI_restore 関数を組にして使用します。

HWI_disable を呼び出すと、アプリケーション内で割り込みがグローバルにディセーブルにされます。C6000 プラットフォームの場合は、HWI_disable はコントロール・ステータス・レジスタ（CSR）の GIE ビットをクリアします。C5000 および C2800 プラットフォームの場合は、HWI_disable は ST1 レジスタの INTM ビットをセットします。どちらのプラットフォームの場合も、これによって、CPU がマスカブル・ハードウェア割り込みを実行できないようになります。したがって、ハードウェア割り込みはグローバル・ベースで機能するので、割り込みイネーブル・レジスタの個々のビットではなく、すべての割り込みに影響を与えます。割り込みを再びイネーブルにするには、HWI_enable または HWI_restore を呼び出します。HWI_enable は C6000 プラットフォームでは GIE ビットを常にセットし、C5000 および C2800 プラットフォームでは ST1 レジスタ の INTM ビットをクリアします。これに対して、HWI_restore はこの値を HWI_disable が呼び出される前の状態に復元します。

4.2.3 DSP/BIOS のリアルタイム・モードのエミュレーションによる影響

TI のエミュレーションは、2 つのデバッグ実行制御モードをサポートしています。

- ストップ・モード
- リアルタイム・モード

ストップ・モードでは、プログラムの実行を完全に制御でき、すべての割り込みをディセーブルにすることができます。リアルタイム・モードでは、他のコードの実行が停止している間にタイムクリティカルな割り込みサービス・ルーチンを実行することができます。2 つの実行モードでは、ソフトウェアによるブレークポイント命令の実行、指定したプログラム空間やデータ空間へのアクセスなどのブレーク・イベントでプログラム実行を中断することができます。

リアルタイム・モードでは、タイムクリティカルな割り込みサービス・ルーチン（フォアグラウンド・コードとも呼ばれる）が継続して実行されている間、バックグラウンドで実行されるコードはブレーク・イベントで中断されます。

4.2.3.1 リアルタイム・モードでの C28x の割り込み動作

C28x のリアルタイム・モードは、次のように 3 種類の状態として定義されています

- デバッグ停止状態

- ❑ 単一命令状態
- ❑ 実行状態

デバッグ停止状態：これは、ソフトウェア・ブレークポイント命令のデコードまたは解析ブレークポイント / ウォッチポイントの発生やホスト・プロセッサからの要求などのブレーク・イベントから入る状態です。

停止すると、タイムクリティカルな割り込みが処理されます。IER および DBGIER レジスタで割り込みがイネーブルにされる場合、割り込みはタイムクリティカルな割り込み / リアルタイムの割り込みと定義されます。この場合、INTM ビットが無視されることに注意してください。

しかし、DBGM ビットを使用して、特定のコード領域で CPU が停止状態に入らないようにする（または、デバッグ・アクセスを実行できないようにする）ことができます。INTM と DBGM を同時に使用すると、特定のコード領域へのすべての割り込みを禁止することができます。指定したコード領域では、デバッガによるレジスタ / メモリの更新が行われないことになります。

```
SETC INTM, DEGM
/ Uninterruptable, unhaltable region of code
CLRC INTM, DBGM
```

ブレークポイントがリアルタイムに存在する場合、CPU が停止して、デバッグ停止モードに入る原因になります。これは、停止モードにおけるブレークポイントの動作と同じです。ソフトウェア・ブレークポイントが元の命令を置き換えるので、ソフトウェア・ブレークポイントの実行を安全に無視したり遅延させたりすることはできないことに注意してください。それ以外の場合、目的の命令セットを実行してはいけません。しかし、CPU が停止する原因となる他の方法が遅延します。ソフトウェア・ブレークポイントを置き換えることは、停止させるべき場所を正確に把握している「意図的な行動」であることに注意することが重要です。一方、（CCS 停止コマンドまたは他のトリガとなるイベントなど）他の方法で停止させると、多くの場合、ユーザにはプログラム実行中に停止した場所はわかりません。

ユーザは、割り込みまたは停止が禁止されている場所にブレークポイントを配置してはいけません。しかし、CPU が割り込み不可で停止できないコード領域にあるときに、CCS 停止コマンドやウォッチポイントに基づいて停止を開始することができます。このような場合、DBGM がセットされない場所に来るまで停止は延期されることになります。これは割り込みも同様で、DBGM がセットされない場所に来るまで遅延が発生することになります

たとえば、Semaphore という変数があって、ISR でインクリメントされ、メイン・ループでデクリメントされると仮定します。割り込みとデバッグのアクセスが処理されるので、両方とも次のイタリック表記された領域では発生しません。

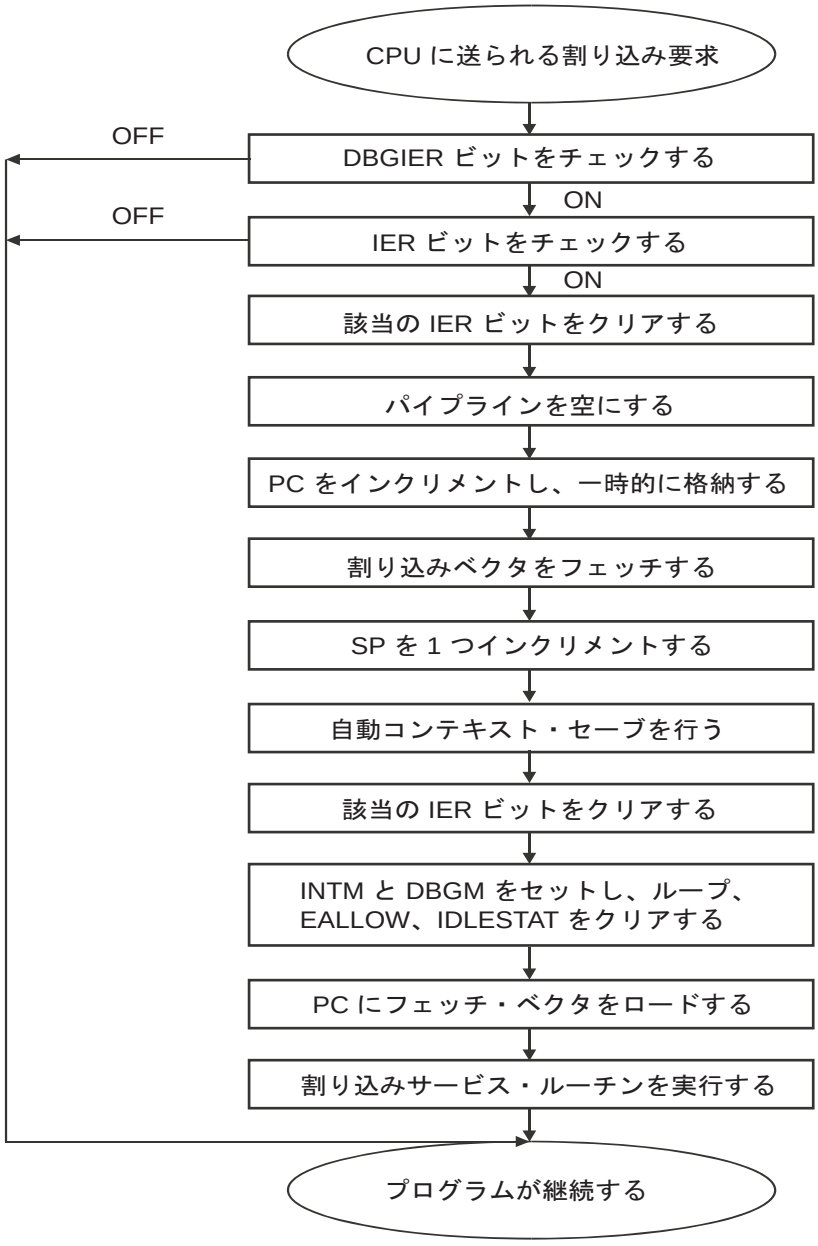
例 4-1.　リアルタイム・モードでの C28x の割り込み動作

```
MAIN_LOOP:
; Do some stuff

SETC INTM, DBGM
/ Uninterruptible, unhaltable region of code
MOV ACC, @Semaphore
SUB ACC, #1    ;Let's do "*Semaphore--;" really inefficiently!
MOV @Semaphore, ACC
CLRC INTM, DBGM

; Do some more stuff
B MAIN_LOOP

; By default, INTM and DBGM are set in an ISR so you can't halt
or interrupt
RT_ISR:

; Do some stuff
MOV ACC, @Semaphore
ADD ACC, #1   ;Let's do "*Semaphore--;" really inefficiently!
MOV @Semaphore, ACC
; Do some more stuff
IRET
```

注：

上記のコードは、デバッガが停止を発行する場合は安全なコードです。ただし、上記のイタリック表記された領域では停止することができないので、PC は常に B MAIN_LOOP 命令にあることになります。ユーザがアドレス Semaphore をアクセスするタイミングで発生するようにウォッチポイントをセットしている場合、CPU は「CLRC INTM, DBGM」の実行が終了するまで停止できません。ユーザが RT_ISR でハードウェア・ブレークポイントをセットすると、同じ結果が発生します。ユーザが上記のイタリック表記された領域にブレークポイントをセットすると、CPU は停止しますがデバッガはこれをエラーとしてレポートし、これが不適切なオペレーションであることを示します。この場合、DEC や INC など極小的な C28x 命令が使用されています。

図 4-3.　デバッグ停止状態の割り込みシーケンス



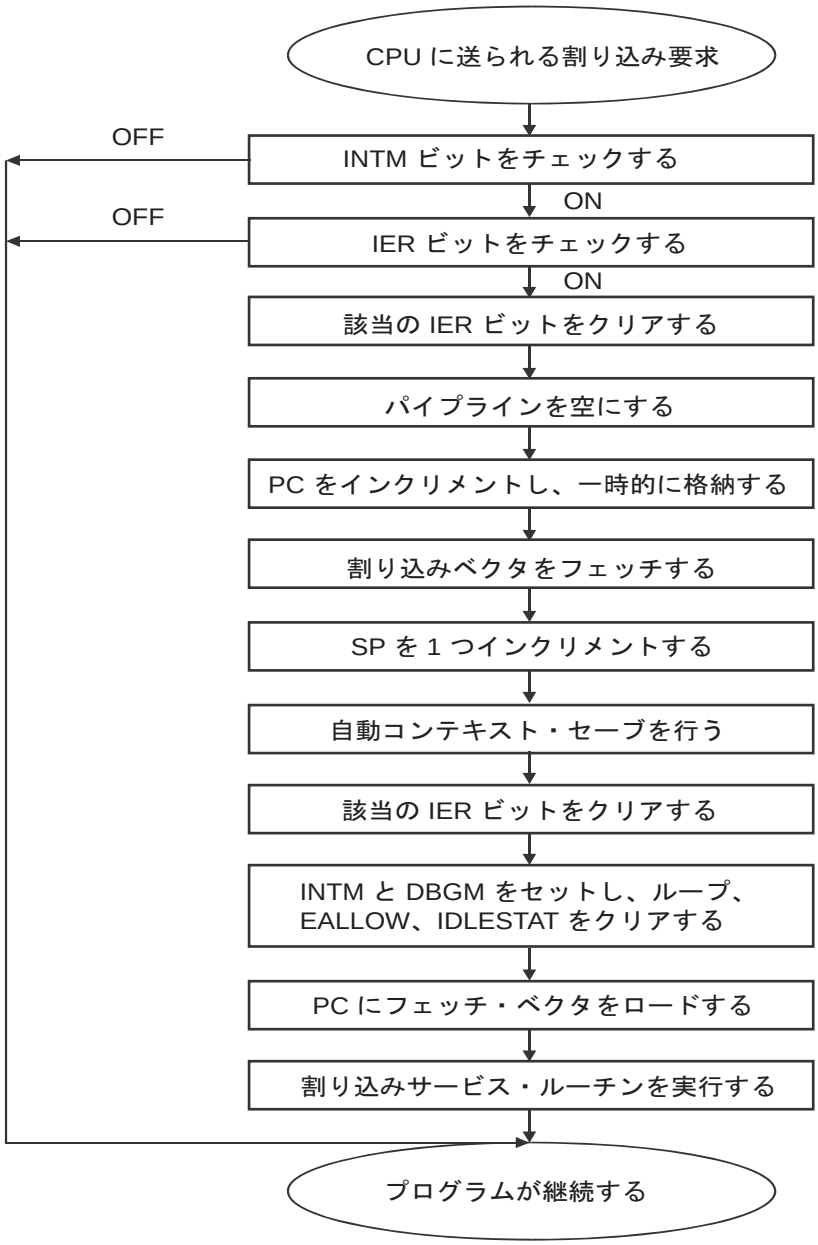
単一命令状態：これは、RUN 1 または STEP 1 コマンドを使用して、デバッガに単一命令を実行するように指示したときに入る状態です。CPU は PC が指し示す単一命令を実行してから、デバッグ停止状態に復帰します。割り込みがこの状態で発生し、かつ RUN 1 コマンドを使用して現在の状態に入った場合、CPU はその割り込みを処理できます。しかし、STEP 1 を使用して現在の状態に入った場合、CPU はその割り込みを処理できません。これは、停止モードおよびリアルタイム・モードの両方にあてはまります。

INTM はシングル・ステップ実行していると見なされていると仮定しても安全であることに注意してください。また、前述の例でコードをシングル・ステップ実行している場合、すべての割り込み不可で停止できないコードは次のように「1 つの命令」として実行されます。

```
PC は最初ここで -> SETC INTM, DBGM
; Uninterruptible, unhaltable region of code
MOV ACC, @Semaphore
SUB ACC, #1   ;Let's do "*Semaphore--;" really inefficiently!
MOV @Semaphore, ACC
CLRC INTM, DBGM

; Do some more stuff
PC はここで停止 -> B MAIN_LOOP
```

実行状態：これは、デバッガ・インターフェイスから run コマンドを使用したときに入る状態です。CPU は、INTM ビットおよび IER レジスタの値に基づいて、すべての割り込みを処理します。

図 4-4.　実行時状態の割り込みシーケンス



DSP/BIOS には、割り込みから保護する必要のあるコード・セグメントが含まれています。このようなコード・セクションはクリティカル・セクションと呼ばれています。このようなセグメントに割り込み、かつ割り込みが一部の DSP/BIOS API を呼び出すと、プログラム結果にエラーが発生します。したがって、「SET INTM, DBGM」と「CLRC INTM, DBGM」の間のコードは重要です。

図 4-2 に、すべての割り込みから保護されている領域の 2 つのコード例を示します。

例 4-2.　割り込みが不可になっているコード領域

(a) アセンブリ・コード

```
.include hwi.h54
...
HWI_disable A ; disable all interrupts, save the old intm
value in reg A
  'do some critical operation'
HWI_restore A0
```

(b) C コード

```
.include hwi.h
Uns oldmask;

oldmask = HWI_disable();
  'do some critical operation; '
  'do not call TSK_sleep(), SEM_post, etc.'
HWI_restore(oldmask);
```

HWI_enable の代わりに HWI_restore を使用すると、呼び出しのペアをネストできます。呼び出しをネストした場合、最も外側の HWI_disable 呼び出しは割り込みをオフにし、最も内側の HWI_disable 呼び出しは何もしません。最も外側の HWI_restore 呼び出しに達するまで、割り込みは再びイネーブルにされません。HWI_enable 呼び出しは、HWI_disable を呼び出したときにすでにディセーブルにされている割り込みもイネーブルにするので、注意してください。

注：

HWI_disable と HWI_enable で囲まれたブロックの中では、タスクの再スケジューリングの原因となるような DSP/BIOS カーネル呼び出し（SEM_post や TSK_sleep など）は使用しないでください。

4.2.4 割り込みの中のコンテキストと割り込み管理

ハードウェア割り込みが現在実行中の関数の優先権を奪う場合、HWI 関数は、使用または変更するレジスタをセーブしたりリストアしたりする必要があります。DSP/BIOS には、レジスタをセーブするための HWI_enter アセンブリ・マクロ、およびレジスタをリストアするための HWI_exit アセンブリ・マクロがあります。これらのマクロを使用すると、優先権を奪われた関数の実行が再開されるときに、その関数に同じコンテキストが与えられます。HWI_enter/HWI_exit マクロは、レジスタ・コンテキストのセーブ / リストア機能に加えて、次のようなシステム・レベルの処理を実行します。

- ❏ SWI および TSK スケジューラが適切な時点で呼び出されるようにする。
- ❏ ISR の実行中に個々の割り込みをディセーブル / リストアする。

HWI_enter アセンブリ・マクロを呼び出す場合、ソフトウェア割り込みまたはセマフォをポストする（またはそれに影響を与える）可能性のあるすべての DSP/BIOS API コールより前に呼び出さなければなりません。HWI_exit アセンブリ・マクロは、関数のコードの最後で呼び出されなければなりません。

全体が C のみで記述された割り込みルーチンをサポートするために、DSP/BIOS は、割り込みルーチン用のこのような入口および出口マクロを実行する HWI ディスパッチャを提供しています。HWI は、この HWI ディスパッチャを使用するかまたは HWI_enter および HWI_exit を明示的に呼び出すことにより、コンテキストのセーブと割り込みのディセーブリングを処理できます。「HWI 構成」プロパティでは、個々の HWI オブジェクトに HWI ディスパッチャを使用するかどうか選択できます。割り込みを取り扱うには、HWI ディスパッチャを使用することをお勧めします。

HWI ディスパッチャは実際には、HWI_enter/HWI_exit マクロのペア内部にある構成済みの HWI 関数を呼び出します。だからこそ完全に C だけで記述された HWI 関数でも取り扱えるのです。ディスパッチャが実際に HWI_enter/HWI_exit マクロのペアが含まれている関数を呼び出したとすれば、システムがクラッシュすることになりかねません。したがって、ディスパッチャを使用する場合は、HWI_enter と HWI_exit のコードのインスタンスを 1 つだけ使用できることになります。

注 :

HWI オブジェクトを C 関数と合わせて使用する場合は、*interrupt* キーワードまたは INTERRUPT プラグマを使用してはいけません。HWI_enter/HWI_exit マクロおよび HWI ディスパッチャにはこの機能が含まれているので、C 修飾子を使用すると破滅的な結果を招く可能性があります。

明示的に呼び出された場合（C55 の場合）も、C55 HWI ディスパッチャで呼び出された場合も、HWI_enter および HWI_exit マクロは、C 関数を呼び出すことができるように ISR を準備します。特に、ISR は、HWI のコンテキストから呼び出すことが可能な DSP/BIOS API 関数ならどれでも呼び出すことができるように準備されます（このような関数の完全なリストについては、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「Functions Callable by Tasks, SWI Handlers, or Hardware ISRs」を参照してください）。

注：

C6000 および C54x プラットフォームでシステム HWI ディスパッチャを使用するときは、HWI 関数で HWI_enter と HWI_exit を呼び出してはいけません。

どの HWI ディスパッチ手法を使用するかに関係なく、DSP/BIOS は、SWI と HWI のどちらの実行時にもシステム・スタックを使用します。システム内に TSK タスクがない場合は、すべてのスレッドがこのシステム・スタックを使用します。TSK タスクがある場合は、各タスクがそれぞれ専用スタックを使用します。タスクが SWI または HWI により優先権を奪われた場合、割り込みスレッドの実行中に DSP/BIOS はシステム・スタックを使用します。

C54x プラットフォームでは、HWI_enter と HWI_exit はどちらも 2 つのパラメータを使用します。

- 第 1 のパラメータは MASK です。ISR がどの CPU レジスタをセーブしリストアするか指定します。
- C54x プラットフォームでは、HWI_enter および HWI_exit の第 2 のパラメータは IMRDISABLEMASK です。HWI_enter マクロ呼び出しと HWI_exit マクロ呼び出しとの間にディセーブルにされる割り込みをマスクします。

  割り込みがトリガされると、プロセッサはグローバルに割り込みを（ステータス・レジスタ ST1 の INTM ビットをセットして）ディセーブルにし、その後、割り込みベクタ・テーブル内に設定されている ISR にジャンプします。HWI_enter マクロは、ST1 レジスタの INTM ビットをクリアすることにより割り込みを再びイネーブルにします。その前に HWI_enter は、インタラプト・マスク・レジスタ（IMR）の該当ビットをクリアすることにより、一部の割り込みを選択的にディセーブルにします。IMR レジスタのどのビットがクリアされているかは、HWI_enter マクロに渡される IMRDISABLEMASK 入力パラメータにより判別されます。したがって、HWI_enter を使用して、現在の HWI 関数の優先権を奪うことができる割り込みと奪うことができない割り込みの選別を制御することができます。

HWI_exit を呼び出すときには、IMRRESTOREMASK パラメータも渡すことができます。IMRRESTOREMASK のビット・パターンにより、HWI_exit がどの割り込みをリストアするか（IMR のどのビットをセットするか）が決まります。IMRRESTOREMASK で指示されている割り込みのうち、HWI_exit がリストアするのは、HWI_enter によりディセーブルにされている割り込みだけです。ISR を終了するときに、HWI_enter によりディセーブルにされた割り込みのいずれかをリストアしたくない場合は、IMRRESTOREMASK のその割り込みに該当するビットを HWI_exit でセットしないようにします。HWI_exit は、IMRRESTOREMASK にはない割り込みビットのステータスには影響を与えません。

C55x プラットフォームでは、全部で 7 つのパラメータを使用できます。最初の 5 つのパラメータはどの CPU レジスタをコンテキストとしてセーブするかを指定するもので、残りの 2 つのパラメータは 2 つの割り込みマスク・ビットマップを指定するために使用できます。

C6000 プラットフォームでは、HWI_enter と HWI_exit はどちらも 4 つのパラメータを使用します。

- 最初の 2 つのパラメータは ABMASK と CMASK です。ISR がどの A、B、および制御レジスタをセーブしリストアするかを指定します。
- C6000 プラットフォームの 3 番目のパラメータは IEMASK です。HWI_enter マクロ呼び出しと HWI_exit マクロ呼び出しの間にディセーブルにされる割り込みをマスクします。

  割り込みがトリガされると、プロセッサはグローバルに割り込みをディセーブルにし（コントロール・ステータス・レジスタ（CSR）の GIE ビットをクリア）、そして割り込みサービス・テーブル内で設定されている ISR にジャンプします。HWI_enter マクロは、CSR の GIE をセットすることにより、割り込みを再びイネーブルにします。その前に HWI_enter は、IEMASK パラメータに指定されているインタラプト・イネーブル・レジスタ（IER）のビットを選択的にディセーブルにします。したがって、HWI_enter を使用して、現在の HWI 関数の優先権を奪うことができる割り込みと奪うことができない割り込みの選別を制御することができます。

  HWI_exit が呼び出されたとき、IEMASK のビット・パターンによって、HWI_exit がどの割り込みをリストアするか（IER のどのビットをセットするか）が決まります。IEMASK で指示されている割り込みのうち、HWI_exit がリストアするのは、HWI_enter によりディセーブルにされている割り込みだけです。ISR を終了するときに、HWI_enter によりディセーブルにされた割り込みのいずれかをリストアしたくない場合は、IEMASK のその割り込みに該当するビットを HWI_exit でセットしないようにします。HWI_exit は、IEMASK にはない割り込みビットのステータスには影響を与えません。

- C6000 プラットフォームの 4 番目のパラメータは CCMASK です。CSR のキャッシュ制御フィールドに入れる値を指定します。このキャッシュ状況は、HWI_enter 呼び出しから HWI_exit 呼び出しまでのコードを実行中効力を維持します。このマスクの代表的な値のいくつかが、c62.h62 の中で定義されています（たとえば、C62_PCC_ENABLE）。PCC コードと DCC コードを OR 演算して、CCMASK を生成することができます。CCMASK に 0 を使用した場合はデフォルト値が使用されます。この値をセットするには、Tconf 構成時に「GBL」プロパティを使用します。

  CLK_F_isr（クロック・マネージャがイネーブルにされているときにオンデバイス・タイマ割り込みの 1 つを操作する）も、構成時にセットされたキャッシュ値を使用します。HWI_enter は、CCMASK により定義されたキャッシュ・ビットをセットする前に、現在の CSR ステータスをセーブします。HWI_exit は、CSR を割り込みコンテキストの時点の値にリストアします。

事前定義されたマスク C62_ABTEMPS および C62_CTEMPS（C62x）または C64_ABTEMPS および C64_CTEMPS（C64x）は、それぞれ、C 言語のすべての一時 A/B レジスタ、すべての一時制御レジスタを指定します。これらのマスクは、C 関数により自由に使用できるレジスタをセーブするために使用できます。C6000 プラットフォームで HWI ディスパッチャを使用しているときはレジスタ・セットを指定することはできないので、これらのマスクにより指定されたすべてのレジスタがセーブされリストアされます。

たとえば、HWI 関数が C 関数を呼び出す場合は次のコードを使用します。

```
HWI_enter C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK
 `isr code`
HWI_exit C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK
```

C 関数または DSP/BIOS 関数のいずれも呼び出していない段階で、HWI_enter を使用して、すべての C ランタイム環境のレジスタをセーブするようにします。これらのレジスタをリストアするには、HWI_exit を使用します。

HWI_enter および HWI_exit は、C ランタイム環境のレジスタをセーブしリストアする他に、割り込みのネストが生じている場合に、最も外側の割り込みだけが DSP/BIOS スケジューラを呼び出すようにします。HWI または他のネストされた HWI が、SWI_post を使用して SWI ハンドラをトリガするか、または優先順位の高いタスクをレディ状態にした場合（たとえば、SEM_ipost または TSK_itick を呼び出すことによって）は、最も外側の HWI_exit が SWI スケジューラと TSK スケジューラを起動します。SWI スケジューラは、優先順位の高いタスクへのコンテキスト切り換えを行う前に、保留状態にあるすべての SWI ハンドラに対するサービスを行います（必要な場合）。

C2800 プラットフォームでは、HWI_enter と HWI_exit はどちらも 4 つのパラメータを使用します。

- 第 1 のパラメータは AR_MASK です。ISR がどの CPU レジスタ（xar0 ～ xar7）をセーブしリストアするか指定します。

- C28x プラットフォームでは、HWI_enter および HWI_exit の第 2 のパラメータは、ACC_MASK です。ISR がストアしリストアする ACC、p、t の各レジスタのマスクを指定します。
- 第 3 のパラメータは、MISC_MASK です。ier、ifr、DBGIER、st0、st1、dp の各レジスタのマスクを指定します。
- 第 4 のパラメータは、IERDISABLEMASK です。IER のどのビットをオフにするか指定します。

割り込みがトリガされると、プロセッサは IER ビットをオフにし、グローバルに割り込みを（ステータス・レジスタ ST1 の INTM ビットをセットして）ディセーブルにし、その後、割り込みベクタ・テーブル内に設定されている ISR にジャンプします。HWI_enter マクロは、ST1 レジスタの INTM ビットをクリアすることにより割り込みを再びイネーブルにします。その前に HWI_enter は、インタラプト・イネーブル・レジスタ（IER）の該当ビットをクリアすることにより、一部の割り込みを選択的にディセーブルにします。IER レジスタのどのビットがクリアされているかは、HWI_enter マクロに渡される IERDISABLEMASK 入力パラメータにより判別されます。したがって、HWI_enter を使用して、現在の HWI 関数の優先権を奪うことができる割り込みと奪うことができない割り込みの選別を制御することができます。HWI_exit を呼び出すときには、IERRESTOREMASK パラメータも渡すことができます。IERRESTOREMASK のビット・パターンにより、HWI_exit がどの割り込みをリストアするか(IER のどのビットをセットするか)が決まります。IERRESTOREMASK で指示されている割り込みのうち、HWI_exit がリストアするのは、HWI_enter によりディセーブルにされている割り込みだけです。ISR を終了するときに、HWI_enter によりディセーブルにされた割り込みのいずれかをリストアしたくない場合は、IERRESTOREMASK のその割り込みに該当するビットを HWI_exit でセットしないようにします。HWI_exit は、IERRESTOREMASK にはない割り込みビットのステータスには影響を与えません。

ISR が呼び出すことができる関数の完全なリストについては、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「Functions Callable by Tasks, SWI Handlers, or Hardware ISRs」を参照してください。

注：

DSP/BIOS 関数を参照する DSP/BIOS アセンブリまたは C 言語の HWI の中のすべての文を、HWI_enter と HWI_exit で囲む必要があります。HWI ディスパッチャを使用すれば、この要件が満たされます。

例 4-3 に、ユーザが HWI ディスパッチャを使用しないことを選択した場合に、C6000 プラットフォームで最小限の HWI を作成するためのアセンブリ言語コードを示します。また、例 4-4 に C54x プラットフォームの場合のコード例、例 4-5 に C55x の場合の例を示します。これらの例では、より事細かな制御を可能にするために HWI_enter を使用しています。

例 4-3.　C6000 プラットフォームで最小限の ISR を組み立てる方法


```
;
; ======== myclk.s62 ========
;
    .include "hwi.h62"  ; macro header file

IEMASK    .set 0
CCMASK    .set c62_PCC_DISABLE
    .text

;
; ======== myclkisr ========
;
    global _myclkisr
_myclkisr:

    ; save all C run-time environment registers
    HWI_enter C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK

    b       _TSK_itick   ; call TSK itick (C function)
    mvkl    tiret, b3
    mvkh    tiret, b3

    nop     3

tiret:

    ; restore saved registers and call DSP/BIOS scheduler
    HWI_exit C62_ABTEMPS, C62_CTEMPS, IEMASK, CCMASK


    .end
```

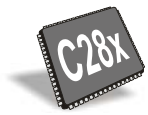
例 4-4.　C54x プラットフォームでの HWI の例


```
;
;  ======== _DSS_isr ========
;
; Calls the C ISR code after setting cpl
; and saving C54_CNOTPRESERVED
;

    .include "hwi.h54"  ; macro header file

_DSS_isr:
    HWI_enter    C54_CNOTPRESERVED, 0fff7h
    ; cpl = 0
    ; dp = GBL_A_SYSPAGE
    ; We need to set cpl bit when going to C
    ssbx    cpl
    nop                 ; cpl latency
    nop                 ; cpl latency
    call    _DSS_cisr
    rsbx    cpl             ; HWI_exit precondition
    nop                 ; cpl latency
    nop                 ; cpl latency
    ld    #GBL_A_SYSPAGE, dp
    HWI_exit    C54_CNOTPRESERVED, 0fff7h
```

例 4-5.　C55x プラットフォームでの HWI の例


```
;
;  ======== _DSS_isr ========
;
_DSS_isr:
    HWI_enter C55_AR_T_SAVE_BY_CALLER_MASK,
              C55_ACC_SAVE_BY_CALLER_MASK,
              C55_MISC1_SAVE_BY_CALLER_MASK,
              C55_MISC2_SAVE_BY_CALLER_MASK,
              C55_MISC3_SAVE_BY_CALLER_MASK,
              0FFF7h,0
              ; macro has ensured 'C' convention,
              ; including SP alignment!
    call    _DSS_cisr
    HWI_exit  C55_AR_T_SAVE_BY_CALLER_MASK,
              C55_ACC_SAVE_BY_CALLER_MASK,
              C55_MISC1_SAVE_BY_CALLER_MASK,
              C55_MISC2_SAVE_BY_CALLER_MASK,
              C55_MISC3_SAVE_BY_CALLER_MASK,
              0FFF7h,0
```

例 4-6.　C28x プラットフォームでの HWI の例




```
;
;  ======== _DSS_isr ========
;
_DSS_isr:
    HWI_enter    AR_MASK,ACC_MASK,MISC_MASK,IERDISABLEMASK
    lcr          _DSS_cisr
    HWI_exit     AR_MASK,ACC_MASK,MISC_MASK,IERDISABLEMASK
```

4.2.5　レジスタ

C 関数によりセーブおよびリストアされる DSP/BIOS レジスタは、標準の C コンパイラ・コードに準拠しています。当社の C ランタイム・モデルに準拠する TMS320 関数によりセーブおよびリストアされるレジスタについては、ご使用のプラットフォームに対応した『オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル』を参照してください。

4.3 ソフトウェア割り込み

ソフトウェア割り込みは、ハードウェア ISR に似ています。DSP/BIOS の SWI モジュールは、ソフトウェア割り込み機能を提供します。ソフトウェア割り込みは、SWI_post などの DSP/BIOS API を呼び出すことにより、プログラムに基づいてトリガされます。ソフトウェア割り込みの優先順位は、タスクよりは高くハードウェア割り込みよりは低くなります。

SWI モジュールを、多くのプロセッサに存在する「SWI」命令と混同しないようにしてください。DSP/BIOS SWI モジュールは、あらゆるプロセッサ固有のソフトウェア割り込み機能とは別に働きます。

SWI スレッドは、比較的遅い速度で発生するアプリケーション・タスク、またはハードウェア割り込みほどリアルタイム・デッドラインが厳しくないアプリケーション・タスクに適しています。

ソフトウェア割り込みをトリガまたはポストできる DSP/BIOS API は、次のとおりです。

- ❏ SWI_andn
- ❏ SWI_dec
- ❏ SWI_inc
- ❏ SWI_or
- ❏ SWI_post

SWI マネージャは、すべてのソフトウェア割り込みの実行を制御します。アプリケーションが上記の API のいずれかを呼び出すと、SWI マネージャは、そのソフトウェア割り込みに対応する関数を実行するためにスケジュールします。SWI マネージャは、SWI オブジェクトを使用して 1 つのアプリケーションに含まれるすべてのソフトウェア割り込みを取り扱います。

ポストされたソフトウェア割り込みは、保留されているすべてのハードウェア割り込みの実行が完了してから実行されます。HWI は、実行中の SWI ルーチンの優先権をいつでも奪うことができます。そして、HWI が完了してから SWI ハンドラが再開されます。一方、SWI ハンドラは常にタスクより優先して実行されます。保留されているすべてのソフトウェア割り込みの実行が完了してから、最も優先順位の高いタスクが実行可能になります。実質的には、SWI ハンドラはすべての通常タスクより優先順位の高いタスクと考えることができます。

注 :

SWI については、注意しなければならないことが 2 つあります。

SWI ハンドラは、ハードウェア割り込みにより割り込まれるか優先順位の高い SWI により優先権を奪われない限り、完了するまで実行されます。

HWI ISR の中で呼び出す場合、ソフトウェア割り込みをトリガまたはポストする可能性のある任意の SWI 関数を呼び出すコード・シーケンスは、HWI_enter/HWI_exit のペアで囲むか、または HWI ディスパッチャにより起動する必要があります。

4.3.1 SWI オブジェクトを作成する方法

他の多くの DSP/BIOS オブジェクトと同様に、SWI オブジェクトも、(SWI_create を呼び出して) 動的に作成したり、(構成時に) 静的に作成したりすることができます。動的に作成した割り込みは、プログラム実行中に削除することもできます。

新しいソフトウェア割り込みを構成に追加するには、Tconf スクリプトに新しい SWI オブジェクトを作成します。アプリケーションが SWI オブジェクトをトリガしたときに、各ソフトウェア割り込みにより実行される関数のプロパティをセットすることができます。それぞれの SWI 関数に渡される引数を最大 2 つまで構成することもできます。

SWI オブジェクトをどのメモリ・セグメントから割り当てるかを決定することができます。ソフトウェア割り込みがポストされ、実行対象としてスケジュールされると、SWI マネージャは SWI オブジェクトにアクセスします。

SWI オブジェクトとそのプロパティに関する説明は、DSP/BIOS オンライン・ヘルプに含まれています。SWI モジュールの API コールの詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「SWI Module」を参照してください。

ソフトウェア割り込みを動的に作成するには、次の構文を使用します。

```
swi = SWI_create(attrs);
```

ここで、swi は割り込みハンドルを、変数 attrs は SWI 属性を指します。SWI 属性の構造体 (SWI_Attrs 型のもの) には、SWI 用に静的に構成できるすべての要素が含まれています。変数 attrs は NULL でも構いませんが、その場合はデフォルトの属性セットが使用されます。一般的には、変数 attrs には少なくともハンドラ用の関数が 1 つ含まれています。

注 :

SWI_create は、HWI や 別の SWI からではなく、タスク・レベルからのみ呼び出すことができます。

SWI_getattrs を使用すると、すべての SWI_Attrs 属性を検索できます。これらの属性の一部はプログラム実行中にも変更できますが、一般的には、これらの属性にはオブジェクト作成時に割り当てられた値が入っています。

```
SWI_getattrs(swi, attrs);
```

4.3.2 ソフトウェア割り込みの優先順位をセットする方法

ソフトウェア割り込みには、さまざまな優先順位レベルがあります。メモリの制約が許す限り、優先順位の各レベルについて、いくつでもソフトウェア割り込みを作成できます。リアルタイム・デッドラインの短いスレッドを取り扱うソフトウェア割り込みには高い優先順位を選択し、実行デッドラインがあまり厳しくないスレッドを取り扱うソフトウェア割り込みには、低い優先順位を選択することができます。

ソフトウェア割り込みには、最大 15 個の優先順位レベルがあります。最高レベルは SWI_MAXPRI (14) です。最低レベルは SWI_MINPRI (0) です。優先順位レベル 0 は、タスク・スケジューラを実行する KNL_swi オブジェクト用として予約されています。スタック・サイズの制限については、4.3.3 節「ソフトウェア割り込みの優先順位とアプリケーションのスタック・サイズ」（4-28 ページ）を参照してください。1 つの優先順位レベルの中でソフトウェア割り込みをソートすることはできません。

4.3.3 ソフトウェア割り込みの優先順位とアプリケーションのスタック・サイズ

タスクを除き、DSP/BIOS のすべてのスレッドは同じシステム・スタックを使用して実行されます。

システム・スタックは、ソフトウェア割り込みが他のスレッドの優先権を奪ったときにレジスタ・コンテキストを保存します。実行時に発生する可能性のある最大数の優先実行を取り扱えるようにするために、必要なスタック・サイズは、ユーザがソフトウェア割り込み優先順位レベルを追加するたびに増加します。したがってスタック・サイズの観点から言えば、各ソフトウェア割り込みにそれぞれ異なる優先順位を割り当てるより、複数のソフトウェア割り込みに同じ優先順位レベルを割り当てる方が効率的です。

MEM モジュールのデフォルトのシステム・スタック・サイズは、256 ワードです。このサイズは構成時に変更できます。構成ツールの上部にあるステータス・バーには、予測される必要サイズが示されます。

最大 15 個のソフトウェア割り込み優先順位レベルを使用できますが、各レベルではより大きなシステム・スタックが必要になります。「the system stack size is too small to support a new software interrupt priority level（システム・スタック・サイズが小さすぎて、新しいソフトウェア割り込み優先順位レベルをサポートできません）」というポップアップ・メッセージが表示されたら、MEM セクション・マネージャの「Application Stack Size」プロパティの値を大きくしてください。

最初の PRD オブジェクトを作成すると、PRD_swi という名前の新しい SWI オブジェクトが作成されます（PRD の詳細については、4.10 節「周期関数マネージャ（PRD）とシステム・クロック」（4-77 ページ）を参照）。最初の PRD オブジェクトの前に SWI オブジェクトが 1 つも作成されていない場合は、PRD_swi を追加すると最初の優先順位レベルが使用され、それに応じて必要なシステム・スタックのサイズが増加します。

TSK マネージャがイネーブルにされている場合は、（KNL_swi という名前の SWI オブジェクトにより実行される）TSK スケジューラは最も低い SWI 優先順位レベルを予約します。他の SWI オブジェクトは、その優先順位をもつことができなくなります。

4.3.4 ソフトウェア割り込みの実行

ソフトウェア割り込みを実行対象としてスケジュールするには、SWI_andn、SWI_dec、SWI_inc、SWI_or、および SWI_post を呼び出します。これらの呼び出しは、割り込みサービス・ルーチン、周期関数、アイドル関数、またはその他のソフトウェア割り込み関数など、事実上プログラム内のどこでも使用することができます。

SWI オブジェクトがポストされると、SWI マネージャは、実行保留状態にあるポスト済みソフトウェア割り込みのリストにそのオブジェクトを追加します。次に、SWI マネージャは、ソフトウェア割り込みが現在イネーブルにされているかどうかチェックします。HWI 関数が実行中であるなどでイネーブルにされていない場合、SWI マネージャは現在のスレッドに制御を返します。

ソフトウェア割り込みがイネーブルにされている場合、SWI マネージャは、ポストされた SWI オブジェクトの優先順位を現在実行中のスレッドの優先順位と比較します。現在実行中のスレッドが、バックグラウンド・アイドル・ループか優先順位の低い SWI である場合、SWI マネージャはポスト済み SWI オブジェクトのリストからその SWI を削除し、CPU 制御を現在のスレッドから切り換えて、ポストされた SWI 関数の実行を開始します。

現在実行中のスレッドが同等またはそれより優先順位の高い SWI である場合、SWI マネージャは現在のスレッドに制御を返し、ポストされた SWI 関数は、それ以前にポストされていて同等以上の優先順位をもつ他のすべての SWI の実行が完了した後に実行されます。

注 :

SWI については、注意しなければならないことが 2 つあります。

SWI は、いったん実行が開始された後は、完了するまでブロックされずに実行されなければなりません。

HWI の中から呼び出す場合、ソフトウェア割り込みをトリガまたはポストする可能性のある、任意の SWI 関数を呼び出すコード・シーケンスは、HWI_enter/HWI_exit のペアで囲むか、または HWI ディスパッチャにより起動する必要があります。

SWI 関数は、自身より優先順位の高いスレッド（HWI または自身より優先順位の高い SWI など）により優先権を奪われることがあります。ただし、SWI 関数はブロックできません。何か（特定のデバイスなど）がレディ状態になるのをソフトウェア割り込みが待っているときも、その割り込みを中断することはできません。

ある SWI が、SWI マネージャがそれをポスト済み SWI リストから削除される前に複数回にわたりポストされた場合でも、その SWI 関数は 1 回しか実行されません。これは、CPU が割り込みフラグ・レジスタ内の対応する割り込みフラグ・ビットをクリアする前にハードウェア割り込みが複数回トリガされても、HWI は 1 回しか実行されないのと同じです （実行対象としてスケジュールされる前に複数回ポストされる SWI の取り扱い方法の詳細については、4.3.5 節「SWI オブジェクトのメールボックスを使用する方法」（4-30 ページ）を参照）。

アプリケーションでは、優先順位が同じ SWI ハンドラの呼び出しについては特定の順序を想定してはいけません。しかし、SWI ハンドラは、自身を安全な方法でポストすること（または他の割り込みでポストすること）ができます。複数の SWI ハンドラが保留状態にある場合、それらはすべて最初のタスクが実行される前に呼び出されます。

4.3.5 SWI オブジェクトのメールボックスを使用する方法

個々の SWI オブジェクトについて、C6000 の場合は 32 ビットのメールボックス、C5400 の場合は 16 ビットのメールボックスをそれぞれ 1 つずつもっています。このメールボックスは、ソフトウェア割り込みをポストすべきかどうか判別するために、または SWI 関数の中で評価できる値として使用されます。

SWI_post、SWI_or、SWI_inc は、SWI オブジェクトを無条件にポストします。

- ❏ SWI_post は、ソフトウェア割り込みをポストするために使用されるときに、SWI_post オブジェクトのメールボックスの値を変更しません。
- ❏ SWI_or は、パラメータとして渡されるマスクに従ってメールボックス内のビットをセットしてから、ソフトウェア割り込みをポストします。
- ❏ SWI_inc は、SWI オブジェクトをポストする前に、SWI のメールボックスの値に 1 を加算します。

SWI_andn および SWI_dec は、メールボックスの値が 0 になった場合にのみ SWI オブジェクトをポストします。

- ❏ SWI_andn は、パラメータとして渡されるマスクに従ってメールボックス内のビットをクリアします。
- ❏ SWI_dec は、メールボックスの値を 1 だけ減らします。

表 4-3 に、これらの関数間の相違点をまとめました。

表 4-3. SWI オブジェクト関数間の相違点

操作	メールボックスをビットマスクとして使用	メールボックスをカウンタとして使用	メールボックスを変更しない
常にポスト	SWI_or	SWI_inc	SWI_post
0 になった場合にポスト	SWI_andn	SWI_dec	—

SWI のメールボックスを使用すると、SWI 関数がポストされる条件、およびソフトウェア割り込みがポストされ実行対象としてスケジュールされた後で SWI 関数が実行される回数を厳密に制御できます。

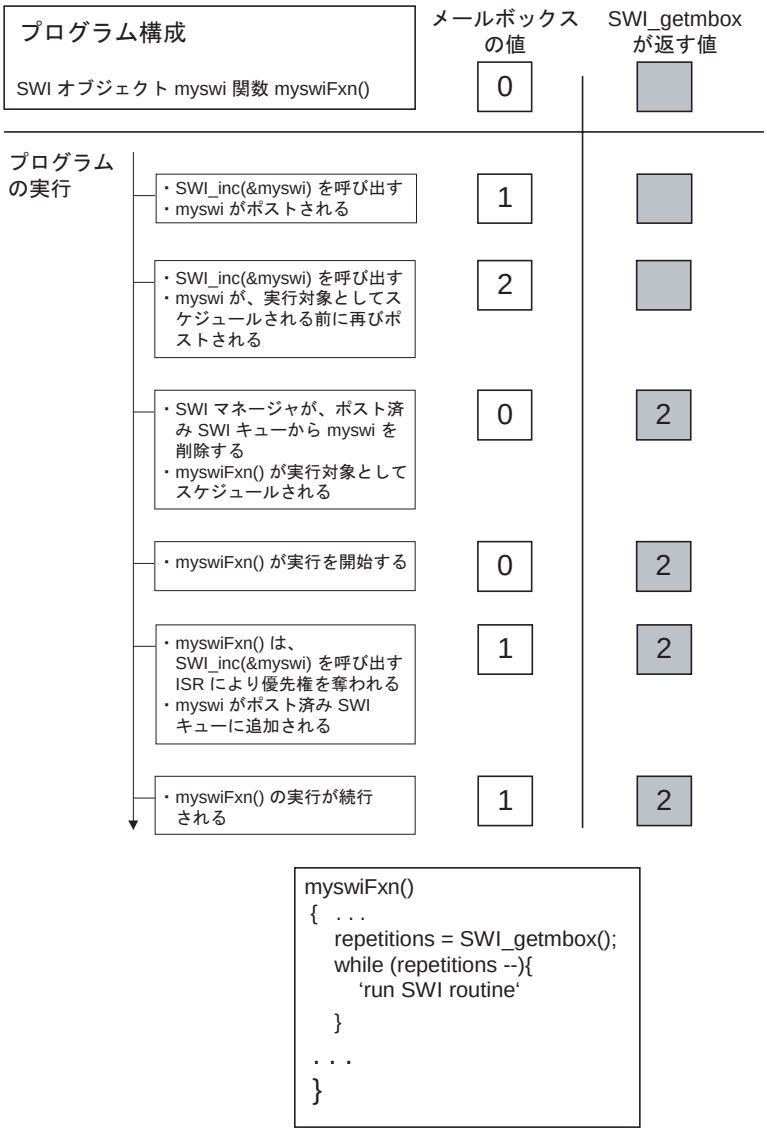
自身のメールボックスの値にアクセスするために、SWI 関数は、SWI_getmbox を呼び出すことことができます。SWI_getmbox は、SWI のオブジェクト関数からのみ呼び出すことができます。SWI_getmbox が返す値は、SWI オブジェクトがポスト済み SWI キューから削除され、SWI 関数を実行対象としてスケジュールする前のメールボックスの値です。

SWI マネージャがポスト済みオブジェクトのキューから保留状態の SWI オブジェクトを削除すると、そのオブジェクトのメールボックスは初期値にリセットされます。メールボックスの初期値は、Tconf スクリプト内でセットしてください。SWI 関数実行中に、その関数が再度ポストされた場合は、それに応じてメールボックスは更新されます。ただし、これによって SWI 関数実行中に SWI_getmbox が返す値が変わることはありません。つまり、SWI_getmbox が返すメールボックスの値は、ソフトウェア割り込みが保留されている SWI のリストから削除された時点の固定的なメールボックスの値です。しかし SWI のメールボックスの値は、SWI が保留されている SWI のリストから削除され実行対象としてスケジュールした直後にリセットされます。これを利用することにより、アプリケーションは、SWI 関数の実行がまだ完了していなくても、新しいポストが発生した場合に SWI のメールボックスの値を更新できます。

たとえば、ある SWI オブジェクトが、ポスト済み SWI のキューから削除される前に複数回ポストされた場合、SWI マネージャがその関数を実行するためにスケジュールするのは 1 回だけです。しかし、SWI オブジェクトが複数回ポストされた場合に SWI 関数も複数回実行されなければならない場合は、図 4-5 に示すように、SWI_inc を使用して SWI をポストする必要があります。

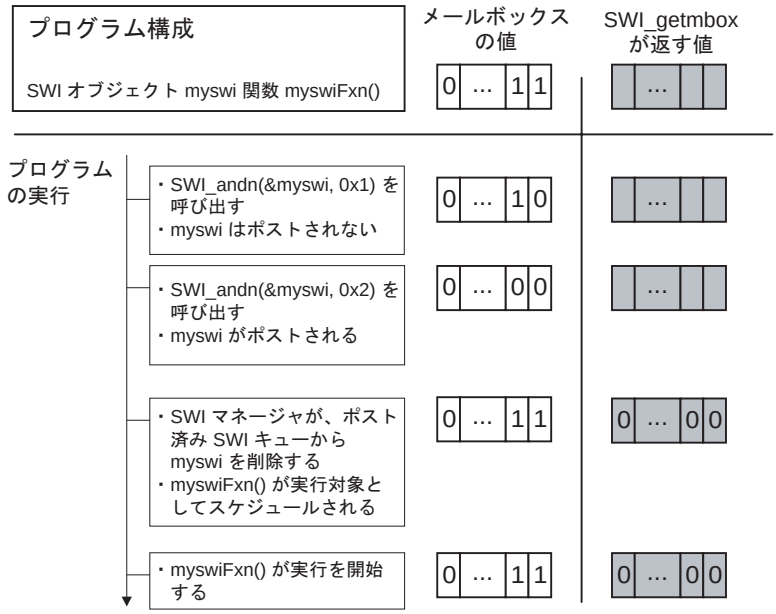
SWI_inc を使用して SWI がポストされた場合、SWI マネージャが対応する SWI 関数が実行できるようにするために呼び出すと、SWI 関数は SWI オブジェクトのメールボックスにアクセスして、そのオブジェクトが実行対象としてスケジュールされる前に何回ポストされたかを判別し、同じルーチンをメールボックスの値と同じ回数だけ実行することができます。

図 4-5.　SWI_inc を使用した SWI のポスト



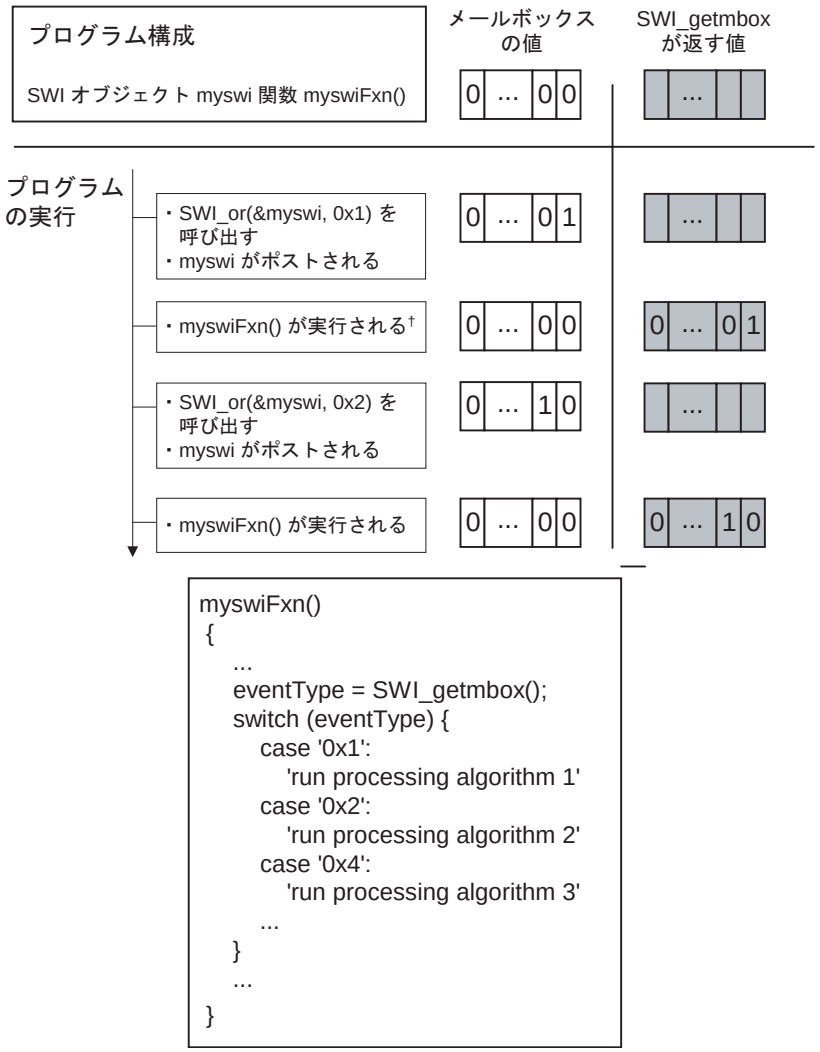
特定のソフトウェア割り込みをトリガするために常に複数のイベントが発生しなければならない場合は、図 4-6 に示すように SWI_andn を使用して該当する SWI オブジェクトをポストします。たとえば、ソフトウェア割り込みが 2 つの異なるデバイスからの入力データを待ってから先へ進まなければならない場合には、SWI オブジェクト構成時に、そのソフトウェア割り込みのメールボックス内で 2 つのビットをセットしておく必要があります。入力データを提供する 2 つのルーチンは、それぞれのタスクを完了したときに、SWI メールボックスのデフォルト値の中でセットされている各ビットをクリアする補数ビットマスクを使用して、SWI_andn を呼び出す必要があります。したがって、両方のプロセスからのデータの準備が整った時点で、初めてソフトウェア割り込みがポストされます。

図 4-6.　SWI_andn を使用した SWI のポスト



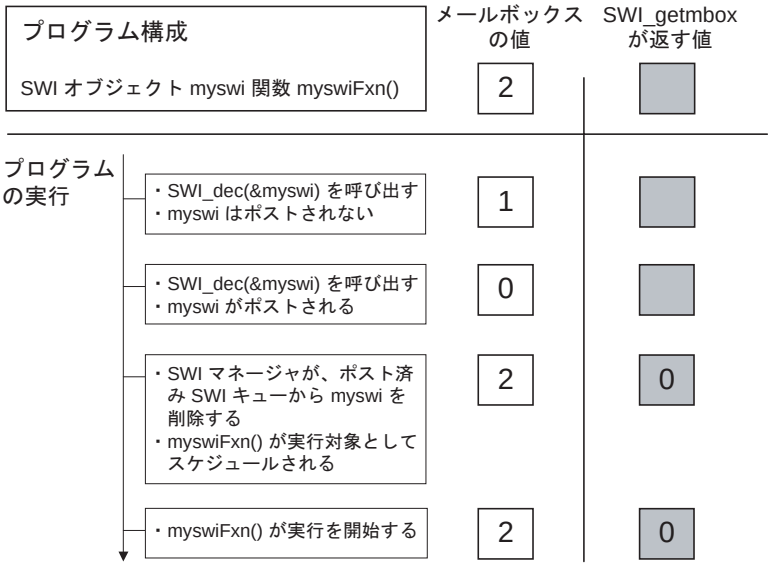
状況によっては、SWI 関数は、どのイベントによってポストされたかに応じて異なるルーチンを呼び出す場合があります。その場合には、プログラムは SWI_or を使用して、何らかのイベントが生じたときに無条件に SWI オブジェクトをポストできます。図 4-7 に、これを示します。SWI_or が使用するビットマスクの値は、ポスト・オペレーションをトリガしたイベントの型をエンコードします。SWI 関数はイベントを識別し、実行するルーチンを選択するためのフラグとしてこの値を使用します。

図 4-7.　SWI_or を使用した SWI のポスト



プログラムを実行するためには、SWI がポストされる前に同じイベントが複数回発生する必要があるという場合は、図 4-8 に示すように SWI_dec を使用して SWI をポストします。SWI メールボックスを、SWI がポストされる前に該当イベントが発生回数と同じになるように構成し、そのイベントが発生するたびに SWI_dec が呼び出されるように構成すれば、メールボックスの値が 0 になった後で、つまりメールボックスの値と同じ回数だけイベントが発生した後で、初めて SWI がポストされます。

図 4-8.　SWI_dec を使用した SWI のポスト



4.3.6　利点と欠点

ハードウェア割り込みではなくソフトウェア割り込みを使用すると、次の 2 つの大きな利点があります。

第 1 に、SWI ハンドラは、すべてのハードウェア割り込みがイネーブルにされた状態のままで実行できます。この利点を明確に認識するために、一般的な HWI はタスクによってもアクセスされるデータ構造を変更するということを思い出してください。したがってタスクは、このようなデータ構造に排他的な方法でアクセスする必要がある場合は、ハードウェア割り込みをディセーブルにしなければなりません。ハードウェア割り込みをディセーブルにすれば、リアルタイム・システムのパフォーマンスが低下するのは明らかです。

これに対して、HWI ではなく SWI ハンドラによって共有データ構造を変更する場合は、タスクが共有データ構造にアクセスしている間はソフトウェア割り込みをディセーブルにすることによって相互排他性を確保できます (SWI_disable と SWI_enable については、本章の後半で説明します)。したがって、システムがハードウェア割り込みを使用して、リアルタイムでイベントに応答する能力が低下することはありません。

多くの場合、長い ISR は 2 つの部分に分割すると効果的です。HWI は非常に時間的重要度の高いオペレーションに専念し、重要度の低い処理は SWI ハンドラに任せます。

第 2 の利点は、SWI ハンドラは、DSP/BIOS が内部データ構造を更新している間は実行されないという保証が得られるため、HWI からは呼び出すことができない一部の関数を SWI ハンドラからは呼び出すことができることです。これは DSP/BIOS の重要な機能の 1 つなので、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「Functions Callable by Tasks, SWI Handlers, or Hardware ISRs」に記述されている表の内容をよく理解しておくようにしてください。この表には、DSP/BIOS 関数と、各関数をどのスレッドから呼び出すことができるか示されています。

注 :

SWI ハンドラは、ブロックの原因にならない DSP/BIOS 関数であればどれでも呼び出すことができます。たとえば、SEM_pend はタスクにブロックを強制することがあるので、SWI ハンドラは SEM_pend も、また SEM_pend を呼び出す関数（MEM_alloc や TSK_sleep など）も呼び出すことはできません。

ただし、SWI ハンドラが完了するまではブロックされているタスクを実行することはできません。状況によっては追加のオーバーヘッドが必要になるとしても、タスクを使用する方が総合的なシステム設計から見れば効果的な場合もあります。

4.3.7 ソフトウェア割り込みの優先権奪取時にレジスタをセーブする方法

ソフトウェア割り込みが他のスレッドの優先権を奪うときには、DSP/BIOS はその優先権を奪われたスレッドのコンテキストを保存するために、表 4-4 に示す、すべての CPU レジスタをシステム・スタックに自動的にセーブします。

表 4-4. ソフトウェア割り込み時にセーブされる CPU レジスタ

C54x プラットフォーム			C55x プラットフォーム			C6000 プラットフォーム		C28x プラットフォーム	
									
ag	ar5	pmst	ac0	rea1	t0	a0-a9	**b16- b31**	al	xt
ah	ar6	rea	ac1	rptc	t1	**a16- a31**	**（C64x のみ）**	ah	ph
al	ar7	rsa	ac2	rsa0	trn1	**（C64x のみ）**		xar0	pl
ar0	bg	sp	ac3	rsa1	xar1		CSR	xar4	dp
ar1	bh	st0	brc1	st0	xar2	b0-99	AMR	xar5	
ar2	bk	st1	brs1	st1	xar3			xar6	
ar3	bl	t	csr	st2	xar4			xar7	
ar4	brc	trn	rea0	st3					

ソフトウェア割り込みが他のスレッドの優先権を奪うときに、表 4-4 に示す、すべてのレジスタは保存されます。C またはアセンブリのいずれかで記述されている SWI ハンドラがこのようなレジスタをセーブする必要はありません。ただし、DSP/BIOS の将来的な実装では、これらのレジスタを保存するとは限らないので、アセンブリで記述された SWI ハンドラは、レジスタ規則に従って「セーブ・オン・エントリ」レジスタを保存する必要があります。このような「セーブ・オン・エントリ」レジスタには、'C54x の場合 ar1、ar6、および ar7、C6000 の場合 a10 ～ a15 および b10 ～ b15 になります （C レジスタ規則の詳細については、ご使用のプラットフォームに対応した『オプティマイジング（最適化）C/C++ コンパイラ ユーザーズ・マニュアル』を参照）。

IER レジスタを変更する SWI 関数は、そのレジスタをセーブし、復帰前にそのレジスタをリストアする必要があります。SWI 関数がこれを行わないと変更は永続的なものになり、新たに実行される他のスレッドや後でプログラムが復帰先のスレッドが、IER に対する変更を受け継ぐことになります。

HWI 関数では、コンテキストは自動的にはセーブされません。ユーザは、HWI_enter マクロと HWI_exit マクロ、または HWI ディスパッチャを使用して、HWI 関数がトリガされたときに割り込まれたコンテキストを保存する必要があります。

4.3.8 SWI ハンドラの同期をとる方法

アイドル・ループ関数、タスク、またはソフトウェア割り込み関数では、すべての SWI 優先権奪取をディセーブルにする SWI_disable を呼び出すことにより、優先順位の高いソフトウェア割り込みによる優先権奪取を一時的に阻止することができます。SWI 優先権奪取を再度イネーブルにするには、SWI_enable を呼び出します。

ソフトウェア割り込みは、グループとしてまとめてイネーブルまたはディセーブルにされます。ソフトウェア割り込みを、個別にイネーブルまたはディセーブルにすることはできません。

DSP/BIOS が初期化を完了すると、最初のタスクが呼び出される前に、ソフトウェア割り込みがイネーブルにされます。アプリケーションは、ソフトウェア割り込みをディセーブルにする必要がある場合は、次のようにして SWI_disable を呼び出します。

```
key = SWI_disable();
```

これに対応するイネーブル関数は、SWI_enable です。

```
SWI_enable(key);
```

key の値は、SWI_disable が複数回呼び出されたかどうかを判別するために SWI モジュールにより使用されます。実際にソフトウェア割り込みをイネーブルにするのは最も外側の SWI_enable 呼び出しだけなので、SWI_disable / SWI_enable 呼び出しをネストすることができます。つまりタスクでは、すでに別のところで SWI_disable が呼び出されているかどうかを判断せずに、ソフトウェア割り込みをディセーブルにしたりイネーブルにしたりできます。

ソフトウェア割り込みがディセーブルにされているときは、その時点ではポスト済みソフトウェア割り込みは実行されません。代わりに、その割り込みはソフトウェア内に「ラッチ」されます。そして、ソフトウェア割り込みがイネーブルにされ、自身が実行可能な優先順位が最も高いスレッドになった時点で実行されます。

注：

SWI_disable には、タスク優先権奪取もディセーブルにされるという重要な副次作用があります。これは、DSP/BIOS がセマフォとクロック・ティックを管理するために、ソフトウェア割り込みを内部的に使用するためです。

動的に作成されたソフトウェア割り込みを削除するには、SWI_delete を使用します。

SWI 用に割り当てられているメモリは解放されます。SWI_delete は、タスク・レベルからのみ呼び出すことができます。

4.4 タスク

DSP/BIOS のタスク・オブジェクトは、TSK モジュールによって管理されているスレッドです。タスクの優先順位は、アイドル・ループより高く、ハードウェア割り込みやソフトウェア割り込みより低くなります。

TSK モジュールは、タスクの優先順位レベルと現在のタスクの実行状態に基づいてタスクを動的にスケジュールし、タスクの優先権を奪います。これにより、プロセッサには常に実行可能な状態になった優先順位が最も高いスレッドが与えられます。タスクには、15 個の優先順位レベルを使用することができます。最も低い優先順位レベル（0）は、アイドル・ループを実行するために予約されています。

TSK モジュールは、タスク・オブジェクトを取り扱う一連の関数を提供します。これらの関数は、TSK_Handle 型のハンドルを使用して TSK オブジェクトにアクセスします。

カーネルは、個々のタスク・オブジェクトに関するプロセッサ・レジスタのコピーを保持しています。各タスクには、ローカル変数を格納するため、および関数呼び出しをさらにネストするための専用のランタイム・スタックがあります。

スタック・サイズは、各 TSK オブジェクトごとに別々に指定できます。各スタックは、通常のサブルーチンの呼び出しだけでなく、1 つ分のタスク優先権奪取コンテキストも取り扱えるだけの十分な大きさを備えている必要があります。タスク優先権奪取コンテキストは、割り込みスレッドにより優先順位の高いタスクがレディ状態にされた結果として、そのタスクが他のタスクの優先権を奪ったときにセーブされるコンテキストです。タスクがブロックされた場合は、C 関数でセーブしなければならないレジスタだけがタスク・スタックにセーブされます。適正なスタック・サイズを判断するには、まず大きいスタック・サイズを設定しておき、Code Composer Studio ソフトウェアを使用して実際に使用されるスタック・サイズを判別します。

1 つのプログラムで実行されるすべてのタスクは、共通のグローバル変数のセットを共有します。これらの変数は、C 関数について定義されている標準の有効範囲規則に従ってアクセスされます。

4.4.1 タスクの作成方法

TSK オブジェクトは、（TSK_create を呼び出して）動的に作成することも、（構成時に）静的に作成することもできます。動的に作成したタスクは、プログラム実行中に削除することもできます。

4.4.1.1 タスクを動的に作成および削除する方法

DSP/BIOS タスクは、TSK_create 関数を呼び出して作成することができます。この関数のパラメータには、新しいタスクの実行を開始する C 関数のアドレスを含めることができます。TSK_create が返す値は TSK_Handle 型のハンドルで、これを他の TSK 関数に引数として渡すことができます。

```
TSK_Handle TSK_create(fxn, attrs, [arg,] ...)
    Fxn        fxn;
    TSK_Attrs  *attrs
    Arg        arg
```

タスクが作成され、かつ、そのタスクが現在実行中のタスクよりも優先順位が高い場合には、実行中のタスクの優先権を奪い、アクティブになります。

TSK オブジェクトと TSK スタックが使用しているメモリは、TSK_delete を呼び出して再利用できます。TSK_delete は MEM_free を呼び出すことにより、すべての内部キューからタスクを削除し、タスク・オブジェクトおよびスタックを解放します。

タスクにより保持されているセマフォ、メールボックス、またはその他のリソースは解放されません。このようなリソースを保持しているタスクを削除するのは、必ずとは言えませんが、アプリケーションの設計上の誤りであることがよくあります。ほとんどの場合、このようなリソースはタスクを削除する前に解放するのが適切です。

```
Void TSK_delete(task)
    TSK_Handle    task;
```

注：

システム内の他のタスクが必要としているリソースを所有しているタスクを削除すると、致命的な障害が生じることがあります。詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「TSK_delete」を参照してください。

4.4.1.2 タスクを静的に作成する方法

Tconf を使用して、タスクを静的に作成することもできます。構成では、各タスクまたは TSK マネージャ自体について、さまざまなプロパティをセットすることができます。「TSK」プロパティの完全な説明については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「TSK Module」を参照してください。

静的に作成したタスクの実行時の動作は TSK_create を使用して作成したタスクと同じです。静的に作成したタスクは、TSK_delete 関数を使用して削除することはできません。静的にオブジェクトを作成する場合の利点については、2.3 節「DSP/BIOS オブジェクトを動的に作成する方法」（2-7 ページ）を参照してください。

デフォルトの構成テンプレートには、優先順位が最も低い TSK_idle タスクが定義されています。このタスクは、自身より優先順位の高いタスク、または割り込みがレディ状態にない場合に、IDL オブジェクト用として定義されている関数を実行します。

注：

DSP/BIOS は、指定されたスタック空間を、ユーザ（データ）スタック・メモリとシステム・スタック・メモリとに等分に分割します。

タスクの優先順位を同じにして構成すると、それらのタスクは構成スクリプトで作成された順序でスケジュールされます。タスクには、最大 16 個の優先順位レベルを使用することができます。最高レベルは 15 で、最低レベルは 0 です。優先順位レベル 0 はシステム・アイドル・タスク用として予約されています。オーダー・プロパティをセットしても、1 つの優先順位レベルの中で、タスクの順位付けをすることはできません。

タスクを最初から中断モードにしたい場合は、その優先順位を -1 にセットします。このようなタスクの実行は、その優先順位が実行時に引き上げられるまでスケジュールされません。

4.4.2 タスクの実行状態とスケジューリング

各 TSK タスク・オブジェクトは、常に次の 4 つの実行状態のうちのいずれかの状態にあります。

1) **実行中（Running）**：タスクがシステムのプロセッサで実際に実行されていることを意味します。

2) **レディ（Ready）**：タスクが、プロセッサの可用性に応じて、実行対象としてスケジュールされることを意味します。

3) **ブロック（Blocked）**：システム内で特定のイベントが発生するまで、タスクを実行できないことを意味します。

4) **終了（Terminated）**：タスクが「終了」していて、再び実行されることはないことを意味します。

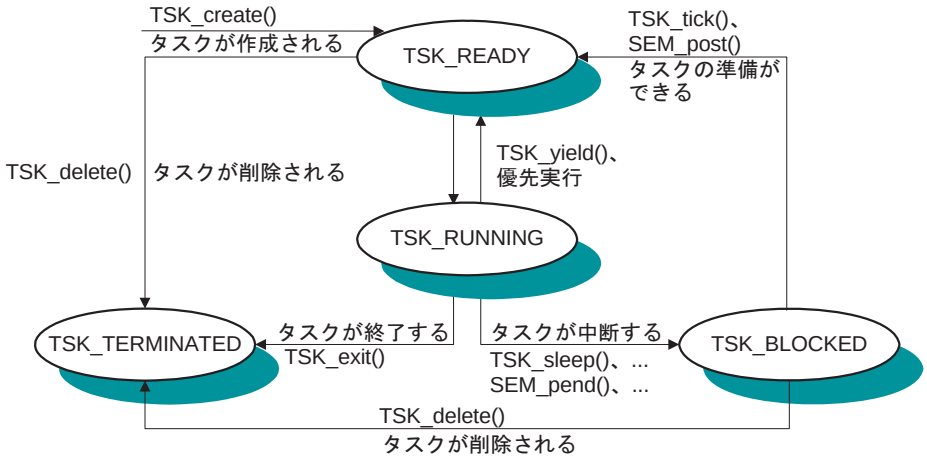
タスクは、アプリケーションに割り当てられている優先順位に従って実行対象としてスケジュールされます。実行中のタスクが同時に複数存在することはできません。原則として、レディ状態のタスクが現在実行中のタスクより高い優先順位レベルをもっていることはありません。TSK は、実行中のタスクより優先順位の高いレディ・タスクがあれば、実行中のタスクの優先権を奪うからです。各タスクにプロセッサの実行時間を「公平に分割する」多くのタイム・シェアリング・オペレーティング・システムの場合と異なり、DSP/BIOS は、現行タスクより優先順位の高いタスクがレディ状態になると直ちに現行タスクの優先権を奪います。

最大優先順位レベルは TSK_MAXPRI (15) で、最低優先順位は TSK_MINPRI (1) です。優先順位が 0 より低いタスクの場合は、後で他のタスクにより優先順位が引き上げられるまでそのタスクの実行は阻止されます。優先順位が TSK_MAXPRI に等しい

場合には、そのタスクを実行すると、ハードウェア割り込みおよびソフトウェア割り込みを処理するだけで他のすべてのプログラム動作は事実上ロックアウトされます。

プログラムの進行中に、いくつかの理由により各タスクの実行モードが変化することがあります。図 4-9 に、実行モードが変化する様子を示します。

図 4-9. 実行モードの変化



タスク・オブジェクトの実行状態は、TSK、SEM、および SIO モジュール内の関数によって、現在実行中のタスクのブロックまたは終了、前に中断されているタスクのレディ、現行タスクの再スケジュールなどに変化します。

実行モードが TSK_RUNNING になっているタスクは、常に 1 個です。すべてのプログラム・タスクがブロックされていて、ハードウェア割り込みもソフトウェア割り込みも実行されていない場合、TSK は、システム内のタスクの中で最も優先順位の低い TSK_idle タスクを実行します。ソフトウェア割り込みやハードウェア割り込みにより優先権を奪われたタスクがあっても、優先権実行が終了すればそのタスクが実行されることになるので、TSK_stat がそのタスクについて返す実行モードは、やはり TSK_RUNNING です。

注：

IDL 関数の中では、SEM_pend や TSK_sleep などブロック用の呼び出しは行わないでください。これを行うと、DSP/BIOS 解析ツールがランタイム情報を収集できなくなります。

TSK_RUNNING タスクが他の 3 つのいずれかの状態に遷移すると、実行可能なレディ状態にある最も優先順位の高いタスク（モードが TSK_READY であるもの）に制御が切り替わります。TSK_RUNNING タスクは、次のようにして他のモードのいずれかに遷移します。

- 実行中のタスクは、TSK_exit が呼び出されると TSK_TERMINATED になります。TSK_exit は、タスクがトップレベル関数から戻ったときに自動的に呼び出されます。すべてのタスクが戻ると、TSK マネージャは、ステータス・コード 0 で SYS_exit を呼び出すことによりプログラムの実行を終了します。

- 実行中のタスクが TSK_BLOCKED になるのは、現行タスクの実行を一時中断させる関数（たとえば、SEM_pend や TSK_sleep）を呼び出した場合です。タスクは、特定の入出力（I/O）操作を行っているとき、何らかの共有リソースが使用可能になるのを待っているとき、またはアイドル状態にあるときに、この状態に移行することがあります。

- タスクは、自身より優先順位の高いタスクが実行可能なレディ状態になると TSK_READY になり、そして優先権を奪われます。TSK_setpri の結果、現行タスクの優先順位がシステム内で最高の順位ではなくなった場合にこのタイプの遷移が生じることがあります。また、タスクは TSK_yield を使用して、同じ優先順位の他のタスクに優先権を譲渡することもできます。譲渡元のタスクはレディ状態になります。

現在 TSK_BLOCKED の状態にあるタスクは、入出力（I/O）操作の完了、共有リソースの可用性、指定した時間の経過など、特定のイベントが発生したときにレディ状態になります。TSK_READY になった結果、このタスクはその優先順位レベルに従って実行対象としてスケジュールされるようになります。そして当然ながら、このタスクの優先順位が現在実行中のタスクより高ければ、このタスクは直ちに実行状態に遷移します。TSK は、同一優先順位のタスクは先入れ先出し方式でスケジュールします。

4.4.3 スタック・オーバーフローをテストする方法

タスクが、スタックに割り当てられているサイズより大量にメモリを使用するときには、他のタスクまたはデータで使用されるメモリ領域に書き込むことができます。これにより予想外の結果、そして場合によっては致命的な結果が生じることがあります。したがって、スタック・オーバーフローをチェックする機能があると便利です。

スタック・サイズを監視するには、TSK_checkstacks と TSK_stat の 2 つの関数を使用できます。TSK_stat が返す構造体には、スタックのサイズ、およびそのスタック内でこれまでに使用された MADU の最大数が含まれているので、次のコード・セグメントを使用してスタックがほぼ満杯になっていることを警告できます。

```
TSK_Stat statbuf;               /* declare buffer */

TSK_stat(TSK_self(), &statbuf); /* call func to get status */
if (statbuf.used > (statbuf.attrs.stacksize * 9 / 10)) {
   LOG_printf(&trace, "Over 90% of task's stack is in use.\n")
}
```

これらの関数の使用方法とその例については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「TSK_stat and TSK_checkstacks」を参照してください。

4.4.4 タスク・フック

アプリケーションは、さまざまなタスク関連のイベントを呼び出すための関数を指定できます。このような関数をフック関数と呼びます。フック関数は、プログラムの初期化、タスク作成（TSK_create）、タスク削除（TSK_delete）、タスク終了（TSK_exit）、タスク準備、タスクのコンテキスト切り換え（TSK_sleep、SEM_pend など）のために呼び出されます。このような関数を使用すると、基本プロセッサ・レジスタ・セットの範囲外にまでタスクのコンテキストを拡張することができます。

TSK モジュール・マネージャでは、フック関数を 1 セット指定することができます。フック関数のセットをさらに作成するには、HOOK モジュールを使用します。たとえば、サードパーティ製のソフトウェアを統合するアプリケーションは、独自のフック関数とサードパーティ製ソフトウェアが必要とするフック関数の両方を実行する必要があります。また、各 HOOK オブジェクトは、各タスクにあわせて専用のデータ環境を保持することができます。

初期の HOOK オブジェクト構成時に、指定した TSK モジュール・フック関数は自動的に HOOK_KNL と呼ばれる HOOK オブジェクトに配置されます。この初期化関数以外のオブジェクトの任意のプロパティをセットするには、「TSK モジュール」プロパティを使用します。HOOK_KNL オブジェクトの初期化関数のプロパティをセットするには、「HOOK オブジェクト」プロパティを使用します。TSK モジュールを使用して、フック関数のセットを 1 つだけ構成する場合、HOOK モジュールは使用されません。

フック関数の詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「TSK Module」および「HOOK Module」を参照してください。

4.4.5 エクストラ・コンテキスト用のタスク・フック

たとえば、タスク単位で保存する必要のある特殊なハードウェア・レジスタ（たとえば拡張アドレッシングのため）を備えたシステムがあるとします。例 4-7 では、doCreate 関数を使用してタスク単位でこれらのレジスタを保持するためにバッファを割り当て、doDelete 関数を使用してこのバッファを解放し、doSwitch 関数を使用してこれらのレジスタをセーブおよびリストアしています。

タスク・オブジェクトを静的に作成すると、Switch 関数は、タスクの環境が例 4-7 のように常に Create 関数によってセットされると見なすことはできません。

例 4-7.　タスク・オブジェクトの作成

```
#define CONTEXTSIZE     `size of additional context`

Void doCreate(task)
    TSK_Handle       task;
{
    Ptr              context;

    context = MEM_alloc(0, CONTEXTSIZE, 0);
    TSK_setenv(task, context);    /* set task environment */
}

Void doDelete(task)
    TSK_Handle       task;
{
    Ptr              context;

    context = TSK_getenv(task);     /* get register buffer */
    MEM_free(0, context, CONTEXTSIZE);
}

Void doSwitch(from, to)
    TSK_Handle       from;
    TSK_Handle       to;
{
    Ptr              context;

    static Int first = TRUE;
    if (first) {
      first = FALSE;
      return;
    }

    context = TSK_getenv(from);      /* get register buffer */
    *context = `hardware registers`; /* save registers */

    context = TSK_getenv(to);        /* get register buffer /
    `hardware registers` = *context; /* restore registers */
}
Void doExit(Void)
{
    TSK_Handle    usrHandle;
    /* get task handle, if needed */
    usrHandle = TSK_self();

    `perform user-defined exit steps`
}
```

注：

LOG_printf に渡す非ポインタ型の関数引数は、次のコード例に示すように明示的に（Arg）に型キャストする必要があります。

```
LOG_printf(&trace, "Task %d Done", (Arg)id);
```

4.4.6 時分割スケジューリングのためのタスク譲渡

例 4-8 に、ユーザが管理できる時分割スケジューリング・モデルの実装例を示しています。このモデルは優先権をもっている（プリエンプティブな）ので、タスクによる連携処理（つまり調整のためのコード）は不要です。タスクは、実行される唯一のスレッドである場合と同様にプログラミングされています。特定のアプリケーション内に、優先順位の異なる複数の DSP/BIOS タスクがあっても構いませんが、時分割モデルが適用されるのは同じ優先順位のタスクだけです。

この例では、prd0 PRD オブジェクトは 1 ミリ秒ごとに TSK_yield() 関数を呼び出す単純な関数を実行するように構成されています。prd1 PRD オブジェクトは 16 ミリ秒ごとに SEM_post(&sem) 関数を呼び出す単純な関数を実行するように構成されています。

図 4-10 に例 4-8 の結果のトレース・ウィンドウを示し、図 4-11 に実行グラフを示します。

例 4-8. 時分割スケジューリング

```
/*
 *  ======== slice.c ========
 *  This example utilizes time-slice scheduling among three
 *  tasks of equal priority. A fourth task of higher
 *  priority periodically preempts execution.
 *
 *  A PRD object drives the time-slice scheduling. Every
 *  millisecond, the PRD object calls TSK_yield()
 *  which forces the current task to relinquish access to
 *  to the CPU. The time slicing could also be driven by
 *  a CLK object (as long as the time slice was the same interval
 *  as the clock interrupt), or by another hardware
 *  interrupt.
 *
 *  The time-slice scheduling is best viewed in the Execution
 *  Graph with SWI logging and PRD logging turned off.
 *
 *  Because a task is always ready to run, this program
 *  does not spend time in the idle loop. Calls to IDL_run()
 *  are added to force the update of the Real-Time Analysis
 *  tools. Calls to IDL_run() are within a TSK_disable(),
 *  TSK_enable() block because the call to IDL_run()
 *  is not reentrant.
 */

#include <std.h>

#include <clk.h>
#include <idl.h>
#include <log.h>
#include <sem.h>
#include <swi.h>
#include <tsk.h>

#include "slicecfg.h"

Void task(Arg id_arg);
Void hi_pri_task(Arg id_arg);
Uns counts_per_us;   /* hardware timer counts per microsecond */
```

例 4-8.　時分割スケジューリング （続き）

```
/*  ======== main ======== */
Void main()
{
    LOG_printf(&trace, "Slice example started!");
    counts_per_us = CLK_countspms() / 1000;
}

/* ======== task ======== */
Void task(Arg id_arg)
{
   Int id = ArgToInt(id_arg);
   LgUns time;
   LgUns prevtime;

   /*
    * The while loop below simulates the work load of
    * the time sharing tasks
    */
   while (1) {
      time = CLK_gethtime() / counts_per_us;

      /* print time only every 200 usec */
      if (time >= prevtime + 200) {
         prevtime = time;
         LOG_printf(&trace, "Task %d: time is(us) Ox%x",
                    id, (Int)time);
      }

      /* check for rollover */
      if (prevtime > time) {
         prevtime = time;
      }

      /*
       * pass through idle loop to pump data to the Real-Time
       * Analysis tools
       */
      TSK_disable();
      IDL_run();
      TSK_enable();
   }
}

/* ======== hi_pri_task ======== */
Void hi_pri_task(Arg id_arg)
{
   Int id = ArgToInt(id_arg);

   while (1) {
      LOG_printf(&trace, "Task %d here", id);

      SEM_pend(&sem, SYS_FOREVER);
   }
}
```

図 4-10.　例 4-8 の結果を示すトレース・ウィンドウ

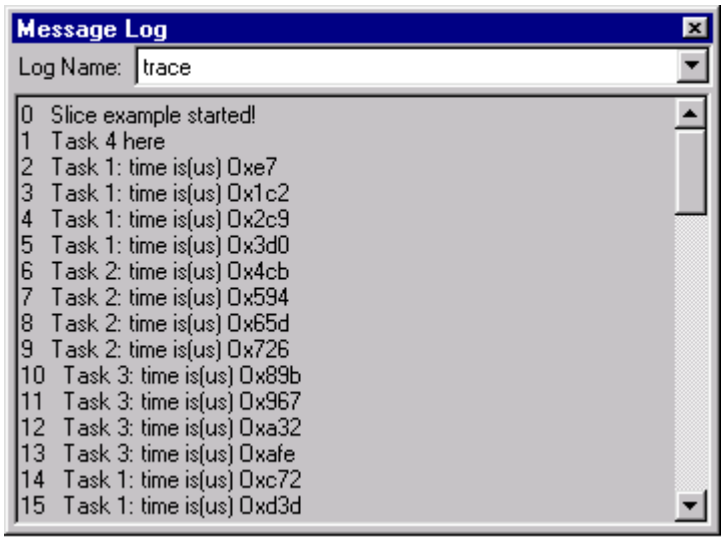
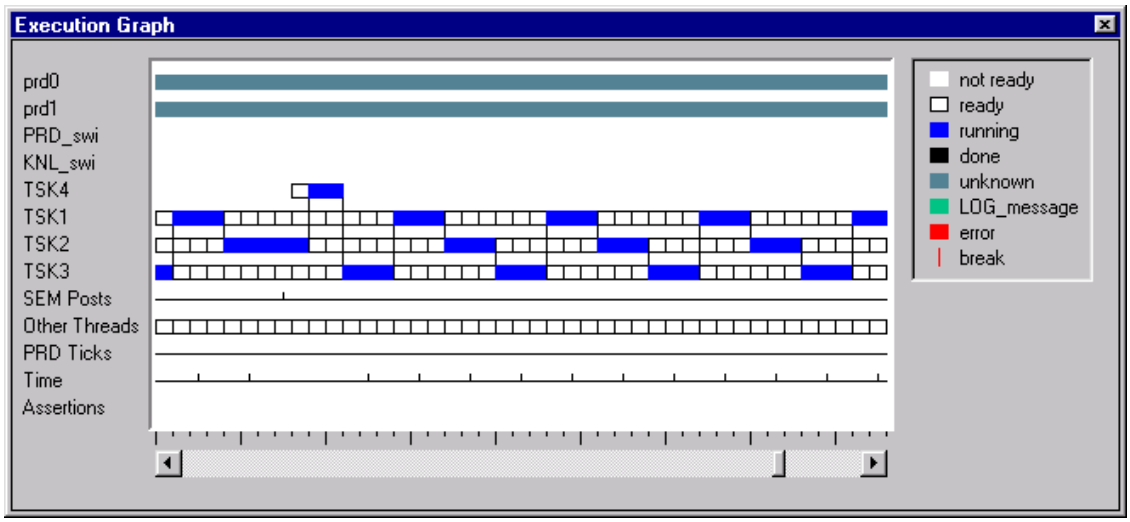


図 4-11.　例 4-8 の実行グラフ



4.5 アイドル・ループ

アイドル・ループは、DSP/BIOS のバックグラウンド・スレッドで、ハードウェア割り込みサービス・ルーチン、ソフトウェア割り込み、またはタスクが 1 つも実行されていないときに継続的に実行されます。その他のスレッドは、どの時点でもアイドル・ループの優先権を奪うことができます。

IDL マネージャを使用すると、アイドル・ループの中で実行する関数を挿入できます。アイドル・ループは、ユーザが構成した IDL 関数を実行します。IDL_loop は、個々の IDL オブジェクトに関連付けられている関数を一度に 1 つずつ呼び出した後、始めに戻って連続ループの形で同じ操作をやり直します。関数は、作成されたときと同じ順序で呼び出されます。したがって、1 つの IDL 関数が完了してからでなければ、その次の IDL 関数の実行を開始することはできません。最後のアイドル関数が完了すると、アイドル・ループは再び最初の IDL 関数を開始します。アイドル・ループ関数は、多くの場合割り込みの生成、システム・ステータスの監視、または他のバックグラウンド動作を行わない（または行うことができない）非リアルタイム・デバイスをポーリングするために使用されます。

アイドル・ループは、DSP/BIOS アプリケーション内で最も優先順位の低いスレッドです。アイドル・ループが実行されるのは、他のハードウェア割り込み、ソフトウェア割り込み、またはタスクを実行する必要がないときだけです。ターゲットと DSP/BIOS 解析ツール間の通信は、バックグラウンドのアイドル・ループにより実行されます。したがって、DSP/BIOS 解析ツールがプログラムの処理を妨げることはありません。ターゲット CPU が使用中でバックグラウンド・プロセスを実行できない場合、DSP/BIOS 解析ツールは、CPU が使用可能になるまでターゲットからの情報の受信を停止します。

デフォルトでは、アイドル・ループは次の IDL オブジェクト用の関数を実行します。

- **LNK_dataPump** は、ターゲット DSP とホスト間のリアルタイム解析データ（LOG データや STS データなど）、および HST チャネル・データの転送を管理します。この処理には RTDX を使用します。

  C54x プラットフォームでは、RTDX_dataPump IDL オブジェクトは RTDX_Poll を呼び出して、ターゲットとホスト間でデータを転送します。これは、最も優先順位が低いアイドル・ループの中で行われます。

  C55x および C6000 プラットフォームでは、ホスト PC が割り込みをトリガしてターゲットとの間でデータを転送します。この割り込みの優先順位は、SWI、TSK、および IDL 関数より高くなっています。実際の HWI 関数は非常に短い時間で実行されます。アイドル・ループの中では、LNK_dataPump 関数は RTDX バッファを準備し、RTDX 呼び出しを実行するという、さらに時間のかかる作業を行います。高い優先順位で実行されるのは、実際のデータ転送だけです。特に大量の LOG データを転送する必要がある場合は、このデータ転送によりわずかながらリアルタイムの動作が影響を受けることがあります。

- **RTA_dispatcher** はターゲット上のリアルタイム解析サーバで、DSP/BIOS 解析ツールからコマンドを受け取り、ターゲットから計測情報を収集し、その情報を実行時にアップロードします。RTA_dispatcher は、2 つの専用 HST チャネルの終端に配置されます。ホストとの間のコマンドまたは応答のやりとりは、LNK_dataPump を介して行われます。

- **IDL_cpuLoad** は、STS オブジェクト（IDL_busyObj）を使用してターゲットの負荷を計算します。このオブジェクトの内容は、CPU の負荷を表示するために、RTA_dispatcher を通して DSP/BIOS 解析ツールにアップロードされます。

- **RTDX_dataPump** は、C5400 プラットフォームでは、RTDX_Poll を呼び出して、ターゲットとホスト間でデータを転送します。これが行われるのは、DSP に IDL ループを定期的に実行できる十分なフリー・サイクルがある場合だけです。C55x および C6000 プラットフォームの場合、RTDX は割り込み駆動型のインターフェイス（LNK_dataPump オブジェクトの説明を参照）なので、RTDX_dataPump オブジェクトはありません。

- **PWRM_idleDomains** は、DSP/BIOS アイドル・ループ内で各種 DSP クロック・ドメインをアイドル状態にする関数を呼び出します。アイドル状態にする各種クロック・ドメインは、PWRM モジュールの構成時に選択することができます。HWI、SWI、TSK のいずれかのスレッドを実行できるようになると、アイドル状態されたクロック・ドメインはそれぞれ前の構成に復元されます。

4.6 電源管理

DSP/BIOS 電源マネージャ PWRM は DSP/BIOS モジュールで、アプリケーションの消費電力を削減する機能があります。現在、PWRM モジュールは 'C5509A EVM で利用できます。他の 'C55x デバイスでもその機能は部分的に利用できます。デバイスごとにサポートされている機能を確認するには、DSP/BIOS リリース・ノートを参照してください。

PWRM モジュールには、次の機能があります。

- **クロック・ドメインをアイドル状態にする機能 :** 特定のクロック・ドメインをアイドル状態にして、アクティブな消費電力を削減することができます。
- **ブート時の低消費電力機能 :** ブート時に自動的に呼び出せる低消費電力機能を指定することができます。この機能は、電力を使用するペリフェラルを必要に応じてアイドル状態にすることができます。
- **DSP デバイス初期化機能 :** PWRM を使用して、ブート時にデバイス固有の低消費電力オペレーションを実行することができます。たとえば、PWRM は通常デフォルトでは必要としないクロック・ドメイン（DMA など）を自動的にアイドル状態にすることができます。また、特定のペリフェラルをアイドル状態に対応させることができるので、そのようなペリフェラルが必要になるまで電力を最も消費しない状態にしておきます。
- **リソースの把握機能 :** ランタイム PWRM API 呼び出しを行って、アプリケーションが依存する特定のリソース（クロック・ドメイン、ペリフェラル、およびクロック・ピンなど）の状況を電源マネージャに通知することができます。必要なリソースの状況がわかれば、PWRM ははっきりとした依存関係がないリソースを積極的にアイドル状態にすることができます。
- **電圧と周波数の測定機能 :** CPU の動作電圧と周波数を動的に変更することができます。これは V/F スケーリングと呼ばれています。電力使用状況は、周波数に直線的に比例していて、電圧に二次曲線的に比例しているので、PWRM モジュールを使用すると、大幅に消費電力を削減することができるようになります。
- **スリープ・モードを使用する機能 :** カスタマイズしたスリープ・モードをセットして、非動作時に低消費電力モードにすることができます。これらは、静的にセットすることも実行時にセットすることもできます。
- **スリープとスケーリングを調整する機能 :** PWRM モジュールが提供する登録と通知のメカニズムを使用して、スリープ・モードと V/F スケーリングを調整することができます。

注 :

PWRM モジュールは、アプリケーションのスケジューリング要件に適合していることを保証していないことに注意してください。このような要件に適合させる責任は、アプリケーションまたは開発者側にあります。

4.6.1 クロック・ドメインをアイドル状態にする機能

TI DSP には、「IDLE」命令が組み込まれていて、アクティブな消費電力を削減するために DSP クロックのゲートをオフにします。これは、実行時に消費電力を削減するために使用する基本メカニズムです。'C55x では、クロックは複数のクロック・ドメイン（CPU、CACHE、DMA、EMIF、PERIPH、CLKGEN）にわかれています。クロック・ドメインは、アイドル・コンフィギュレーション・レジスタ（ICR）の該当ビットをセットしてから、IDLE 命令を実行することでアイドル状態にすることができます。

クロック・ドメインをアイドル状態にするときは、アプリケーション・スケジューリングに悪い影響を与えないように注意が必要です。たとえば、より多くのデータが到着するまで DSP CPU をアイドル状態にするタスクがある場合、優先順位がそのタスク以下の他のタスクは次の割り込みが発生するまで実行できません。このようなタスクは、他のタスクのスケジューリングをブロックしてしまいます。この状態を回避するには、DSP CPU は DSP/BIOS がアイドル・ループにあるときだけアイドル状態になるようにします。つまり、DSP CPU は実行待ちになっているスレッドが他にないときに実行します。

この機能を利用するためには、PWRM モジュールを使用して DSP/BIOS アイドル・ループ内の選択したクロック・ドメインを自動的にアイドル状態にします。クロック・ドメインは、PWRM モジュール・プロパティを使用して静的に構成し、PWRM_configure 関数で動的に変更することができます。

IDL ループ内でクロック・ドメインがアイドル状態になるように PWRM を構成すると、他の IDL ループ処理は以前実行していたほど定常的に実行されません。たとえば、ランタイム解析がイネーブルにされていると、アイドル・ループは CPU の負荷を計算する関数を実行し、ランタイム解析データを DSP から収集し、DSP から Code Composer Studio にデータを送り込みます。PWRM アイドル機能がイネーブルのときは、PWRM_F_idleDomains 関数はアイドル・ループ関数に追加されます。PWRM が CPU ドメインをアイドル状態にすると、アイドル・ループ内で PWRM_F_idleDomains が実行されるたびに、CPU ドメインは次の割り込みが発生するまで中断されます。したがって、Code Composer Studio でのリアルタイム解析データの動的な更新が滞り、「ぎこちない」ように見えます。

アイドル・ループ内でクロック・ドメインをアイドル状態にするのは、運用システムでの使用を考えたものです。つまり、Code Composer Studio を使用しないシステム用です。

PWRM_idleClocks 関数は、クロック・ドメインを迅速にしかもいつまでもアイドル状態にしておく方法を提供します。たとえば、アプリケーションが完全にオンチップ・メモリから実行されている場合、PWRM_idleClocks を呼び出して EMIF クロック・ドメインをアイドル状態にすることができます。

4.6.2 ブート時の消費電力削減機能

通常、DSP は電源が完全に供給されて最大のクロック・レートで起動します。しかし、初期段階では不要な、または特定のアプリケーションでは決して使用されないリソースに当然のように電源が供給されています。

PWRM はフック・メカニズムを提供していて、ブート時に呼び出せる関数を指定して、実際にそのリソースが必要になるまで電源を使用するリソースの電源をオフにしたり、アイドル状態にしたりすることができます。たとえば、アプリケーション実行時に送信回路を後で開けて、組み込まれている物理デバイスの電源を入れることができます。後で送信回路を閉じると、物理デバイスの電源を切ることができます。

ブート関数の中で、外部デバイスにコマンドを発行して低消費電力モードに移行するといったことができます。このような機能をメイン・ルーチンに直接実装できますが、ブート・フック・メカニズムを使用すると電源関連コードを電源マネージャに緊密に関連付けることができます。

4.6.3 電源マネージャによるデバイスの初期化

前項で説明したフック・メカニズムを使用すると、アプリケーションがオーディオ・アンプやラジオのサブシステムなどの外部ペリフェラル・デバイスの電源をブート時にオフにできるようになります。一部の DSP デバイスでは、PWRM は「デバイス初期化」メカニズムを提供します。それによって、PWRM はブート時に DSP デバイスを探して、すべての適切なオンチップ・ペリフェラルやドメインを低消費電力状態にします。アプリケーションを実行すると、ペリフェラルとドメインを必要に応じて起動することできます。PWRM によるデバイス初期化機能は、ON / OFF のいずれかに構成できます。デバイス初期化時の PWRM の動作はデバイスによって異なるので、該当する DSP/BIOS リリース・ノートでご確認ください。

4.6.4 リソースの把握

代表的な DSP/BIOS アプリケーションはさまざまなペリフェラル（タイマ、シリアルポートなど）を使用して、その目的を達成しています。通常、デバイス・ドライバは低レベルのペリフェラル・アクセスを管理していますが、アプリケーションがペリフェラルを直接アクセスする場合もあります。このようなシナリオでは、DSP/BIOS カーネル自体でも使用しているペリフェラルがどれなのか「把握」していません。この情報はアプリケーション・コードやドライバ全体に渡って配布されますが、カーネル内には格納されません。結果的に、DSP/BIOS 電源マネージャはアプリケーションが実際に必要とするリソースが何か常にわからなくなり、アプリケーションを「ブレーク状態にする」可能性のないリソースを積極的にはアイドル状態にすることができません。

一部のデバイスでは、PWRM は「リソース把握」機能を提供して、より積極的に電源管理を可能にしています。アプリケーション、ドライバ、および DSP/BIOS モジュールは、特定のリソースが必要になると、PWRM API を呼び出して、それらのリソースの依存関係を宣言することができます。たとえば、ポートが閉じられたときやオーディオ・ドライバを使う作業がないときなど、リソースが必要ではなくなったとき、PWRM API を使ってリソースの依存関係を解除することができます。PWRM は「セット」と「解除」の呼び出しの回数をカウントして、最初に「セット」オペレーションが発生したときにリソースをオンにします。また、最後に「解除」呼び出しが発生したときに該当リソースをオフにします。

リソースの把握は、PWRM のデバイス初期化機能と組み合わせて使用できるようになっています。

1) ブート時に、PWRM はリソースを初期化して電源をオフにします。

2) アプリケーションでリソース（DMA など）を使用する必要があるときは、PWRM_setDependency を呼び出してその依存関係を登録します。次に、PWRM は自動的にリソースに電源を供給します（DMA クロック・ドメインをアイドル状態から変更するなど）。

3) リソースが不要になったとき、アプリケーションは PWRM_releaseDependency を呼び出します。該当リソースに登録されている依存関係がなければ、PWRM は自動的に電源をオフにします（DMA ドメインをアイドル状態にするなど）。

従来のレガシーなコードがあって、リソースの把握用の PWRM 呼び出しを追加するように簡単には変更できない場合があります。たとえば、ドライバはバイナリ形式でのみ利用できます。このような場合、ブート時に PWRM_setDependency への呼び出しを追加して、レガシーなコードの依存関係をはっきりさせることができます。

PWRM が把握しているリソースはデバイスによって異なるので、該当の DSP/BIOS リリース・ノートでご確認ください。

4.6.5 電圧と周波数の測定機能

CMOS ベースの DSP のアクティブな消費電力は、クロック・レート（周波数）に直線的に比例し、動作電圧に二次曲線的に比例します。また、動作電圧は使用可能な最大クロック・レートを決定します。

したがって、アプリケーションが CPU クロック・レートを削減しても処理デッドラインの条件を満たすことができる場合には、消費電力を直線的に比例して削減することができます。しかし、CPU のクロック・レートを下げると、比例して実行時間が伸びることにもなるので、リアルタイム要件が満たされていることを確実にするために、アプリケーションの解析は注意する必要があります。

クロック周波数が下がり、DSP がサポートしている動作電圧が低くなっても新しい周波数が対応している場合、二次曲線的な関係により電圧を下げることにより、場合によってはさらに大幅に電力を削減できます。

PWRM モジュールを使用すると、アプリケーションは PWRM_changeSetpoint を呼び出して、動作電圧および周波数（V/F）を変更することができます。このため、たとえば、アプリケーションが処理要件の少ないモードに切り替わったとき、電力を使用しない状態にするために段階的に電圧と周波数を下げることができます。あるいは、アプリケーションがデータに依存する処理時の「余分な」時間を累積してから、V/F を下げて低消費電力状態で実行している間にその余分な時間を吸収するといった使い方も可能です。

また、アプリケーションは、プラットフォームでサポートされている PWRM_getCurrentSetpoint、PWRM_getNumSetpoints、PWRM_getSetpointInfo、および PWRM_getTransitionLatency 関数を使用する V/F スケーリング機能について学ぶこともできます。

PWRM モジュールは登録通知メカニズムを通じて、アプリケーション全体に渡り V/F 変更を調整できます。クライアントで V/F スケーリング電力イベントが通知されるように PWRM を登録すると、サポートしている V/F 設定値が示されます。このため、たとえば、ドライバが特定の周波数未満では動作しない場合、クライアントがこの設定値を示すのはクライアントが登録されている限りは、さらに低い周波数への遷移を禁止することを PWRM モジュールで登録したときです。

PWRM モジュールは、プラットフォーム固有のパワー・スケーリング・ライブラリ（PSL）を使用して V/F スケーリングの変更を行います。このライブラリは、特定のプラットフォームの場合にのみ実装されています。パワー・スケーリング・ライブラリについては、『Using the Power Scaling Library on the TMS320C5510』（文献番号 SPRA848）を参照してください。

4.6.5.1 DSP/BIOS CLK モジュールの影響

'C5509A では、V/F スケーリング（CPU）の影響を受けるクロックは、クロック・サービス（CLK モジュール）を行う DSP/BIOS が使用するタイマをドライブするクロックと同じです。これは、V/F 設定値を変更すると DSP/BIOS クロック・サービスが中断されることを意味します。中断時間を最小限に抑えるために、PWRM を使用すると DSP/BIOS CLK モジュールは V/F スケーリング・イベントの通知を登録することができます。新しい V/F 設定値が通知されると、CLK モジュールはスケーリング動作の前に使用される同じレートでティックするようにタイマを再プログラムします。

したがって、低分解能時間（CLK_getltime）は次の周波数スケーリングとして引き続き機能します。しかし、少量の絶対時間は動作を再プログラムするために失われる場合があります。損失が発生するのは、V/F スケーリングが発生する前の最後の段階で DSP/BIOS タイマが一時的に停止するからです。スケーリング動作が発生するとすぐに、スケーリング動作の前に使用される同じレートでタイマはティッキングを開始します。スケーリング動作中、タイマは実質的に DSP/BIOS とそのアプリケーションを表します。クロックが停止している間に失われた時間、タイマが再プログラムされて新しい CPU 周波数を使用して同じレートでティックする間に失われた時間を取り戻す努力は必要ありません。また、絶対的な精度は、新しい入力周波数が適切に分周されて選択したティック・レートを収集できるかによって異なります。

高分解能時間（CLK_gethtime）は、次の警告に従って V/F スケーリングと組み合わせて使用することができます。

- ❏ 設定値の遷移全体に渡り、CLK_gethtime の差を比較すると誤った値が生成されます。設定値が遷移する間に、CLK_gethtime を使用して、高分解能の差を取得することができます。
- ❏ タイマがインクリメントまたはデクリメントするレートは通常、各種 V/F 設定値で異なります。

4.6.6 スリープ・モードを使用する機能

PWRM を使用して、アプリケーションをスリープ・モード、つまり DSP を低消費電力状態にすることができます。スリープ・モードの実装はターゲット・プラットフォームごとに異なり、クロックをアイドル状態にしたり、動作電圧を下げたり、サブシステムの電源をオフにしたりします。現在、'C5509A でサポートされているスリープ・モードは、ディープ・スリープと再起動までスリープの 2 つです。

- **ディープ・スリープ**を使用すると、外部割り込みを待つ間 DSP は最も電力を使用しない状態に入ることができます。割り込みが発生すると、DSP は指定していた時点から処理を適切にかつ迅速に再開します。デフォルトでは、すべてのクロック・ドメインはディープ・スリープ・モードではアイドル状態ですが、PWRM を使用すると、これを変更し、特定のクロック・ドメインが、ディープ・スリープ時にアイドル状態になるように構成することもできます。

- **再起動までスリープ**は、さらに低電力状態にしたモードです。DSP を再起動するまで実行を再開しない最も電力を消費しない状態にしておきます。

PWRM モジュールは登録通知メカニズムを通じて、アプリケーション全体に渡りスリープ状態の変更を調整できます。たとえば、外部コーデックを制御するドライバは、DSP がディープ・スリープ・モードに遷移するときに通知されるように構成できるので、外部デバイスに低消費電力状態に移行するように通知することができます。DSP がディープ・スリープ・モードから起動すると、再びドライバは通知を受け、外部コーデックを起動する適切なコマンドを送ります。

4.6.7 スリープとスケーリングを調整する機能

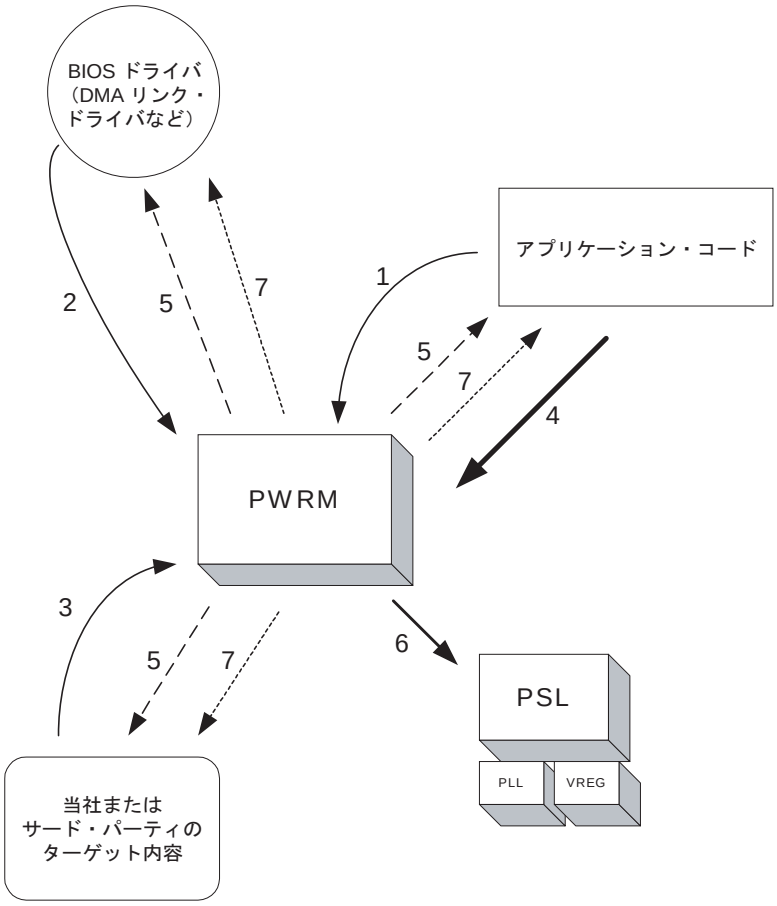
PWRM を使用すると、特定の電力イベントが発生した時点で通知されるように、電力イベントに注意するようなコードを登録することができます。同様に、通知を必要としない場合に、コードの登録を解除することができます。

クライアントが PWRM_registerNotify を呼び出して、次のタイプのイベントについて通知されるように登録することができます。

- 電力イベント
  - V/F 設定値は、これから変更されます。
  - 保留されている V/F 設定値変更は、今行われました。
- スリープ・イベント
  - DSP は、ディープ・スリープ・モードに移行しようとしています。
  - DSP は、ディープ・スリープ・モードから起動しました。
  - DSP は、ディープ・スリープ・モードに移行しようとして、再起動してから再開する必要があります。

図 4-12 に、クライアントが V/F パワー・スケーリング・イベントを登録して通知が行われる様子を示します。

図 4-12.　パワー・イベントの通知



数字が付いた手順は、次のようになります。

1) アプリケーション・コードは、V/F 設定値の変更が通知されるように登録します。異なる設定値に対して異なる EMIF の設定が必要になる場合があります。アプリケーションは設定値が変更されると EMIF の設定が変更できるように電源マネージャ（PWRM）に制御コードを登録します。

2) DMA を使用して外部メモリ・レジスタ間の転送を行う DSP/BIOS リンク・ドライバは V/F 設定値の変更が通知されるように登録します。たとえば、設定値変更の前に、ドライバが外部メモリに対する DMA 操作を一時的に中断することが必要になる場合があります。

3) パッケージ化されたターゲット内容は、同様に設定値が変更になった時点で通知を登録します。

4) アプリケーションは、V/F 設定値の変更を決定し、PWRM_changeSetpoint を呼び出して、設定値の変更を初期化します。たとえば、デバイスのモードを変更したときに実行することが可能です。

5) 設定値変更前に、PWRM は変更要求を検証します。次に行われようとしている設定値変更をすべての登録クライアントに通知します。クライアントには、通知を登録したのと同じ順序（FIFO 順）で通知されます。

6) PWRM は、パワー・スケーリング・ライブラリを呼び出して、V/F 設定値を変更します。

7) 設定値変更後に、PWRM はクライアントに設定値が変更されていることを通知します。

クライアントの通知関数が迅速に実行できる場合、実行し、PWRM_NOTIFYDONE を返します。通知関数が待機する必要があるため実行できない場合、PWRM_NOTIFYNOTDONE を返します。後で、クライアントがその要求動作を完了すると（たとえば、デバイスからの次の割り込み時に）、PWRM_registerNotify が示す delayedCompletionFunction を呼び出します。PWRM モジュールは続行する前に、すべてのクライアントが PWRM_NOTIFYDONE を返すまで待機します。クライアントが指定したタイムアウト内にそれが行われる信号を送らないと、PWRM はシステムに障害があることを示す PWRM_ETIMEOUT を返します。

電力イベントをクライアントに通知する前に、PWRM はまず SWI と TSK のスケジューリングをディセーブルにして、イベントを処理している間に優先実行が行われないようにします。次の表に、イベント処理前後での SWI と TSK のスケジューリングをディセーブルにしたり、再イネーブルにしたりするときの様子を示します。

イベントの種類	SWI と TSK のスケジューリングをディセーブルにする	スケジューリングを再イネーブルにする
V/F スケーリング	クライアントに保留されている設定値変更通知が登録されたことを通知する前。	設定値を変更し、クライアントに完了した設定値変更通知が登録されたことを通知後。
スリープ	クライアントにディープ・スリープへ移行する通知か、再起動までディープ・スリープへ移行する通知が登録されたことを通知する前。	起動して、クライアントにディープ・スリープから起動したという通知が登録されたことを通知後。

PWRM 通知中、SWI と TSK のスケジューリングがディセーブルにされているので、クライアントは PWRM イベント処理を終了させるためにそのスケジューリングに依存できません。HWI を使用して、電力イベントの処理完了を決定して信号を送ることができます。たとえば、インプロセス DMA 動作の完了を可能にするには、DMA ISR が依然として動作していて、delayedCompletionFunction を呼び出して、クライアントがこのイベントの処理を終了したという信号を PWRM に送ります。

4.7 セマフォ

DSP/BIOS は、セマフォに基づいてタスク間の同期と通信を行うための一連の基本的な関数を提供します。セマフォは、一組の互いに競合するタスク間で共有リソースへのアクセスを調整するためによく使用されます。SEM モジュールは、SEM_Handle 型のハンドルを介してアクセスされるセマフォ・オブジェクトを操作する関数を提供します。

SEM オブジェクトは、タスクの同期および相互排他の両方に使用できる計数セマフォ（counting semaphore）です。計数セマフォは、使用可能な対応するリソースの数を示す内部カウントを保持しています。カウントが 0 より大きいときは、タスクはセマフォを獲得しようとしたときにブロックされません。

関数 SEM_create および SEM_delete は、セマフォ・オブジェクトを作成したり削除したりするために使用します（例 4-9 を参照）。また、セマフォ・オブジェクトを静的に作成することもできます。静的にオブジェクトを作成する場合の利点については、2.3 節「DSP/BIOS オブジェクトを動的に作成する方法」（2-7 ページ）を参照してください。

例 4-9. セマフォの作成方法と削除方法

```
SEM_Handle SEM_create(count, attrs);
    Uns        count;
    SEM_Attrs  *attrs;

Void SEM_delete(sem);
    SEM_Handle sem;
```

セマフォ・カウントは、作成時に count に初期化されます。通常、count はセマフォが同期化しているリソース数にセットされます。

SEM_pend はセマフォを待ちます。セマフォ・カウントが 0 より大きい場合、SEM_pend はカウントを減らすだけで返ります。そうでない場合、SEM_pend は、SEM_post がセマフォをポストするまで待ちます。

注：

HWI の中で呼び出される場合、SEM_post または SEM_ipost を呼び出すコード・シーケンスは、一対の HWI_enter/HWI_exit で囲むか、または HWI ディスパッチャにより起動する必要があります。

例 4-10 に示すように SEM_pend で timeout パラメータを使用すると、タスクが timeout まで待つか、無期限に待つか（SYS_FOREVER）、またはまったく待たないか（0）を指定できます。SEM_pend の返り値は、セマフォの獲得に成功したかどうかを示すために使用します。

例 4-10. SEM_pend を使用してタイムアウトを設定する方法

```
Bool SEM_pend(sem, timeout);
  SEM_Handle sem;
  Uns   timeout; /* return after this many system clock ticks*/
```

例 4-11 に示す例では、SEM_post はセマフォをポストするために使用されます。SEM_post は、セマフォを待っているタスクがある場合に、そのタスクをセマフォ・キューから削除し、レディ・キューに入れます。待っているタスクがない場合、SEM_post はセマフォ・カウントを増やすだけで戻ります。

例 4-11. SEM_post を使用してセマフォを通知する方法

```
Void SEM_post(sem);
    SEM_Handle  sem;
```

4.7.1 SEM の例

例 4-12 に、3 つのライタ・タスク用のサンプル・コードを示します。これらのタスクは、それぞれ固有のメッセージを作成して 1 つのリーダ・タスク用のキューに入れます。ライタ・タスクは、新しいメッセージがキューに入れられたことを知らせるために SEM_post を呼び出します。リーダ・タスクは、メッセージを待つために SEM_pend を呼び出します。SEM_pend は、キュー上に使用可能なメッセージがある場合にのみ戻ります。リーダ・タスクは、LOG_printf 関数を使用してメッセージを出力します。

このプログラム例の中の 3 つのライタ・タスク、リーダ・タスク、セマフォ、およびキューは、静的に作成されています。

このプログラムは複数のタスクを使用しているので、キューへのアクセスを同期化するために計数セマフォが使用されています。図 4-13 に、例 4-11 の結果を示します。3 つのライタ・タスクは最初にスケジュールされますが、リーダのタスク優先順位の方がライタより高いので、メッセージはキューに入れられた時点で直ちに読み取られます。

例 4-12.　3 つのライタ・タスクを使用した SEM の例

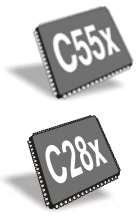
```
  /*
*  ======== semtest.c ========
 *
 *  Use a QUE queue and SEM semaphore to send messages from
 *  multiple writer() tasks to a single reader() task. The
 *  reader task, the three writer tasks, queues, and semaphore
 *  are created statically.
 *
 *  The MsgObj's are preallocated in main(), and put on the
 *  free queue. The writer tasks get free message structures
 *  from the free queue, write the message, and then put the
 *  message structure onto the message queue.
 *  This example builds on quetest.c. The major differences are:
 *    - one reader() and multiple writer() tasks.
 *    - SEM_pend() and SEM_post() are used to synchronize
 *      access to the message queue.
 *    - 'id' field was added to MsgObj to specify writer()
 *      task id.
 *
 *  Unlike a mailbox, a queue can hold an arbitrary number of
 *  messages or elements. Each message must, however, be a
 *  structure with a QUE_Elem as its first field.
 */

#include <std.h>
#include <log.h>
#include <mem.h>
#include <que.h>
#include <sem.h>
#include <sys.h>
#include <tsk.h>
#include <trc.h>

#define NUMMSGS     3 /* number of messages */
#define NUMWRITERS  3 /* number of writer tasks created with */
                      /* Config Tool */

typedef struct MsgObj {
    QUE_Elem    elem;        /* first field for QUE */
    Int         id;          /* writer task id */
    Char        val;         /* message value */
} MsgObj, *Msg;

Void reader();
Void writer();

/*
 *  The following objects are created statically.
 */
extern SEM_Obj sem;

extern QUE_Obj msgQueue;
extern QUE_Obj freeQueue;

extern LOG_Obj trace
```

例 4-12.　3 つのライタ・タスクを使用した SEM の例 （続き）

```
 /*
 *  ======== main ========
 */
Void main()
{
    Int        i;
    MsgObj     *msg;
    Uns        mask;

    mask = TRC_LOGTSK | TRC_LOGSWI | TRC_STSSWI | TRC_LOGCLK;
    TRC_enable(TRC_GBLHOST | TRC_GBLTARG | mask);

    msg = (MsgObj *)MEM_alloc(0, NUMMSGS * sizeof(MsgObj), 0);
    if (msg == MEM_ILLEGAL) {
        SYS_abort("Memory allocation failed!\n");
    }

    /* Put all messages on freequeue */
    for (i = 0; i < NUMMSGS; msg++, i++) {
        QUE_put(&freeQueue, msg);
    }
}

/*
 *  ======== reader ========
 */
Void reader()
{
    Msg         msg;
    Int         i;

    for (i = 0; i < NUMMSGS * NUMWRITERS; i++) {
        /*
         * Wait for semaphore to be posted by writer().
         */
        SEM_pend(&sem, SYS_FOREVER);

        /* dequeue message */
        msg = QUE_get(&msgQueue);

        /* print value */
       LOG_printf(&trace, "read '%c' from (%d).", msg->val, msg-
>id);

        /* free msg */
        QUE_put(&freeQueue, msg);
    }
    LOG_printf(&trace, "reader done.");
}
```

例 4-12.　3 つのライタ・タスクを使用した SEM の例 （続き）

```
 /*
 *  ======== writer ========
 */
Void writer(Int id)
{
    Msg         msg;
    Int         i;

    for (i = 0; i < NUMMSGS; i++) {
        /*
         * Get msg from the free queue. Since reader is higher
         * priority and only blocks on sem, this queue is
         * never empty.
         */
        if (QUE_empty(&freeQueue)) {
            SYS_abort(“Empty free queue!\n”);
        }
        msg = QUE_get(&freeQueue);

        /* fill in value */
        msg->id = id;
        msg->val = (i & 0xf) + ‘a’;
      LOG_printf(&trace, “(%d) writing ‘%c’ ...”, id, msg->val);

        /* enqueue message */
        QUE_put(&msgQueue, msg);

        /* post semaphore */
        SEM_post(&sem);

        /* what happens if you call TSK_yield() here? */
        /* TSK_yield(); */
    }
```

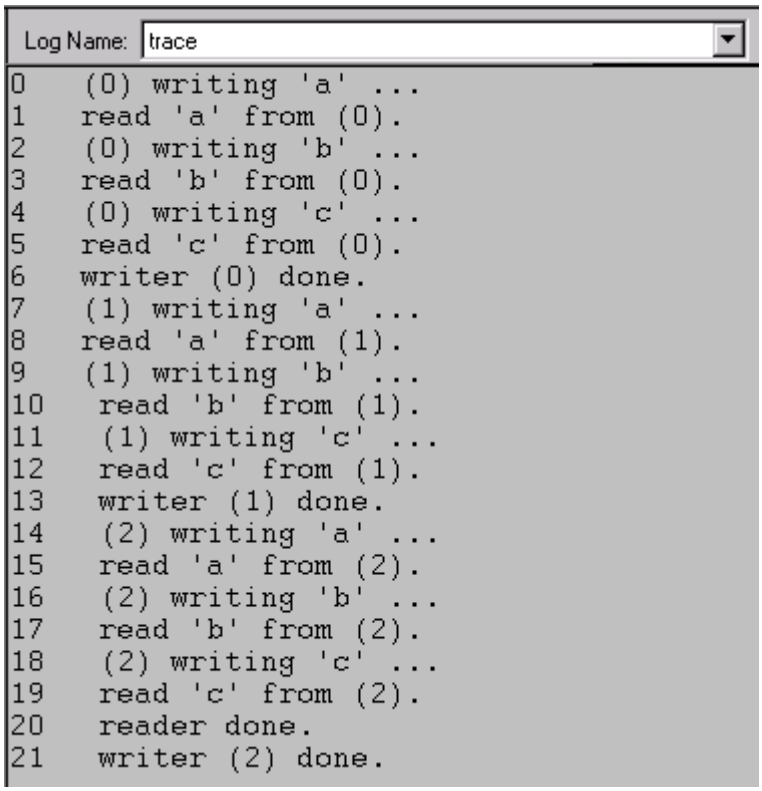


注：

LOG_printf に渡す非ポインタ型の関数引数は、次のコード例に示すように明示的に（Arg）に型キャストする必要があります。

```
LOG_printf(&trace, "Task %d Done", (Arg)id);
```

図 4-13.　例 4-12 の結果を示すトレース・ウィンドウ



4.8 メールボックス

MBX モジュールは、メールボックスを管理するための一連の関数を提供します。MBX メールボックスは、同一プロセッサ上のタスクからタスクへメッセージを渡すために使用できます。固定長の共有メールボックスで強制実行されるタスク間同期の手法を使用すると、着信メッセージのフローがシステムのメッセージ処理能力を超過しないようにすることができます。この節に示す例は、単にこの仕組みを説明するだけのものです。

MBX モジュールが管理するメールボックスは、SWI オブジェクトに含まれているメールボックス構造体とは別のものです。

MBX_create および MBX_delete は、メールボックスを作成したり削除したりするために使用します。また、メールボックス・オブジェクトを静的に作成することもできます。静的にオブジェクトを作成する場合の利点については、2.3 節「DSP/BIOS オブジェクトを動的に作成する方法」（2-7 ページ）を参照してください。

メールボックスを作成するときは、例 4-13 に示すようにメールボックスの長さとメッセージ・サイズを指定します。

例 4-13. メールボックスの作成方法

```
MBX_Handle MBX_create(msgsize, mbxlength, attrs)
    Uns         msgsize;
    Uns         mbxlength;
    MBX_Attrs   *attrs;

Void MBX_delete(mbx)
    MBX_Handle      mbx;
```

MBX_pend は、例 4-14 に示すようにメールボックスからメッセージを読み取るために使用します。メッセージがない場合（つまりメールボックスが空の場合）は、MBX_pend はブロックされます。この場合、タスクは timeout パラメータに従って、タイムアウトまで待つか、無期限に待つか、またはまったく待たないかのいずれかになります。

例 4-14. メールボックスからメッセージを読み取る方法

```
Bool MBX_pend(mbx, msg, timeout)
   MBX_Handle      mbx;
   Void            *msg;
   Uns             timeout;      /* return after this many */
                                 /* system clock ticks */
```

これに対して MBX_post は、例 4-15 に示すように、メールボックスにメッセージをポストするために使用します。使用可能なメッセージ・スロットがない場合（つまりメールボックスが満杯の場合）、MBX_post はブロックされます。この場合、タスクは timeout パラメータに従って、timeout まで待つか、無期限に待つか、またはまったく待たないかのいずれかになります。

例 4-15.　メールボックスへメッセージをポストする方法

```
Bool MBX_post(mbx, msg, timeout)
    MBX_Handle  mbx;
    Void        *msg;
    Uns         timeout;    /* return after this many */
                            /* system clock ticks */
```

4.8.1　MBX の例

例 4-16 のサンプル・コードは、2 つのタイプのタスクが静的に作成されています。メールボックスからメッセージを削除する 1 個のリーダ・タスクと、メールボックスにメッセージを挿入する複数のライタ・タスクです。例 4-16 は、図 4-14 の結果のトレースを示しています。

注 :

HWI の中で呼び出される場合、MBX_post を呼び出すコード・シーケンスは、一対の HWI_enter/HWI_exit で囲むか、または HWI ディスパッチャにより起動する必要があります。

例 4-16.　2 つのタイプのタスクがある MBX の例

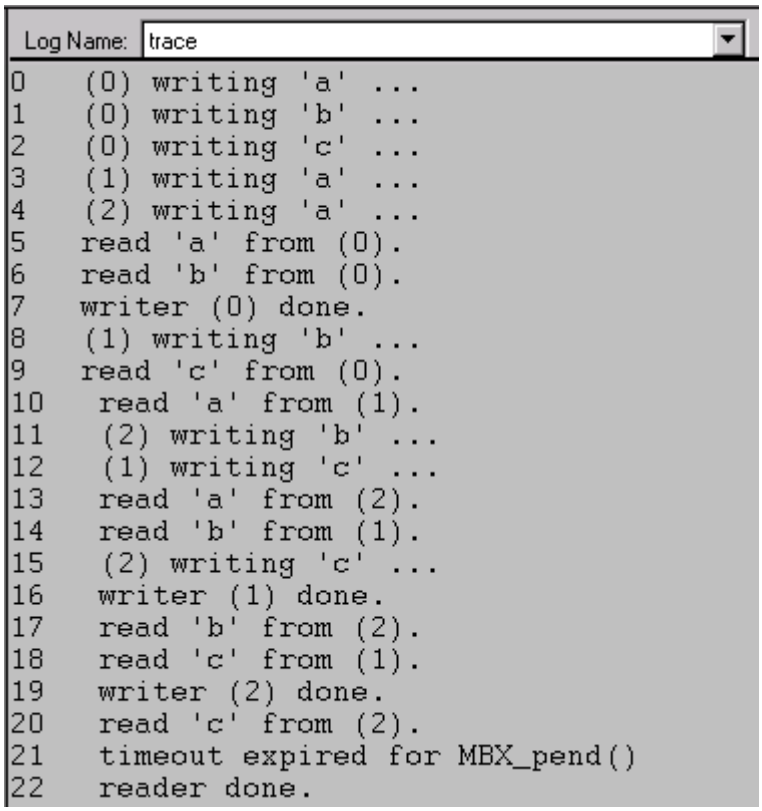
```
/*
 *  ======== mbxtest.c ========
 *  Use a MBX mailbox to send messages from multiple writer()
 *  tasks to a single reader() task.
 *  The mailbox, reader task, and 3 writer tasks are created
 *  statically.
 *
 *  This example is similar to semtest.c. The major differences
 *  are:
 *  - MBX is used in place of QUE and SEM.
 *  - the 'elem' field is removed from MsgObj.
 *  - reader() task is *not* higher priority than writer task.
 *  - reader() looks at return value of MBX_pend() for timeout
 */

#include <std.h>

#include <log.h>
#include <mbx.h>
#include <tsk.h>

#define NUMMSGS         3       /* number of messages */
#define TIMEOUT         10

typedef struct MsgObj {
    Int    id;             /* writer task id */
    Char   val;            /* message value */
} MsgObj, *Msg;

/* Mailbox created with Config Tool */
extern MBX_Obj mbx;

/* "trace" Log created with Config Tool */
extern LOG_Obj trace;

Void reader(Void);
Void writer(Int id);

/*
 *  ======== main ========
 */
Void main()
{
    /* Does nothing */
}
```

例 4-16.　2 つのタイプのタスクがある MBX の例（続き）

```
/*
 *  ======== reader ========
 */
Void reader(Void)
{
    MsgObj      msg;
    Int         i;

    for (i=0; ;i++) {

        /* wait for mailbox to be posted by writer() */
        if (MBX_pend(&mbx, &msg, TIMEOUT) == 0) {
            LOG_printf(&trace, "timeout expired for MBX_pend()");
            break;
        }

        /* print value */
      LOG_printf(&trace, "read 'c' from (d).", msg.val, msg.id);
    }
    LOG_printf(&trace, "reader done.");
}

/*
 *  ======== writer ========
 */
Void writer(Int id)
{
    MsgObj      msg;
    Int         i;

    for (i=0; i < NUMMSGS; i++) {
        /* fill in value */
        msg.id = id;
        msg.val = i % NUMMSGS + (Int)('a');

        LOG_printf(&trace, "(d) writing 'c' ...", id,
(Int)msg.val);

        /* enqueue message */
        MBX_post(&mbx, &msg, TIMEOUT);

        /* what happens if you call TSK_yield() here? */
        /* TSK_yield(); */
    }
    LOG_printf(&trace, "writer (d) done.", id);
}
```

プログラムの実行が完了したら、トレース・ログの内容を調べてください。結果は例 4-14 のようになります。

図 4-14.　例 4-16 の結果を示すトレース・ウィンドウ



メールボックスには、作成時に使用可能なメッセージ・スロットの合計数が指定してあります。これは、メールボックスの作成時にユーザが指定したメールボックスの長さによって決まります。メールボックスへの書き込みを行うタスクを同期化するために計数セマフォが作成され、count がメールボックスの長さにセットされます。タスクが MBX_post 操作を行うと、このカウントは減少します。リーダ・タスクの使用とメールボックスを同期化するために、別のセマフォが作成されます。この計数セマフォは最初ゼロにセットされているので、リーダ・タスクは空のメールボックス上でブロックされます。メールボックスにメッセージがポストされると、このセマフォのカウントは増加します。

例 4-16 では、すべてのタスクの優先順位は同じです。ライタ・タスクはすべてのメッセージをポストしようとしますが、メールボックスが満杯の場合、各ライタは無期限にブロックされます。リーダはその後、空のメールボックスでブロックされるまでメッセージの読み取りを続けます。そして、ライタがすべてのメッセージを供給してしまうまでこのサイクルが繰り返されます。

この時点で、リーダは次の式により求められる時間だけ保留状態になり、その後タイムアウトが生じます。

```
TIMEOUT*1ms/(clock ticks per millisecond)
```

このタイムアウトが生じた後、保留されているリーダ・タスクは実行を続行した後に完了します。

この時点で、スケジューリングと優先順位、参加プログラムの数、メールボックスの長さ、および待ち時間の相対的な影響を、次のようなコード変更を組み合わせながら実際に調べてみると効果的です。

- ❑ タスクの作成順序または優先順位
- ❑ リーダおよびライタの数
- ❑ メールボックスの長さのパラメータ（MBXLENGTH）
- ❑ ライタ・タスクのタイムアウトを扱うコードの追加

4.9 タイマ、割り込み、およびシステム・クロック

通常、DSP は、定期的にハードウェア割り込みを発生させる 1 つまたは複数のオンデバイス・タイマを備えています。DSP/BIOS は一般に、自身のシステム・クロックのソースとして使用可能なオンデバイス・タイマの 1 つを使用します。CLK モジュールは、ほとんどの TMS320 DSP に備わっているオンデバイス・タイマ・ハードウェアを使用して、単一命令サイクルに近い時間分解能をサポートします。

システム・クロックに関するパラメータは、DSP/BIOS 構成設定で定義します。DSP/BIOS システム・クロックの他に、タイマ割り込みが発生するたびに関数を実行するようにするための追加のクロック・オブジェクトを設定できます。

C6000 プラットフォームでは、CLK モジュールの HWI オブジェクトに関するパラメータも定義できます。これは、このオブジェクトは HWI ディスパッチャを使用するように事前構成されているからです。したがって、ユーザは、CLK ISR の割り込みマスクとキャッシュ制御マスクを操作することができます。

DSP/BIOS は、2 つの独立したタイミング方式を提供します。高分解能および低分解能の時間、およびシステム・クロックです。デフォルト構成では、低分解能時間とシステム・クロックは同じです。ただしユーザ・プログラムでは、データの可用性など他のイベントを使用してシステム・クロックを駆動することもできます。ユーザは、CLK マネージャによるオンデバイス・タイマの使用をディセーブルまたはイネーブルにすることにより、高分解能時間および低分解能時間を駆動できます。システム・クロックは、低分解能時間またはその他のイベントを使用して、またはイベントをまったく使用せずに駆動することができます。これらの 2 つのタイミング方式の間の相互作用は、例 4-15 に示します。

図 4-15. 2 つのタイミング方式の間の相互作用

	CLK モジュールが システム・クロックを駆動	他のイベントが システム・クロックを駆動	システム・クロックを 駆動するイベントなし
CLK マネージャは イネーブル	デフォルト構成： 低分解能とシステム・ クロックが同じ	低分解能時間と システム・クロックは 異なる	低分解能時間と高分解能 時間のみ使用可能。 タイムアウトにならない。
CLK マネージャは ディスエーブル	不可能	システム・クロックのみ 使用可能。CLK 関数は 実行されない。	タイミング方式なし。 CLK 関数は実行されず、 タイムアウトにもならない。

4.9.1 高分解能クロックと低分解能クロック

構成時に CLK マネージャを使用すると、「Clock Manager Properties」ダイアログ・ボックスでオンデバイス・タイマを使用して高分解能時間および低分解能時間を駆動する DSP/BIOS の機能をディセーブルまたはイネーブルにできます。

C6000 プラットフォームには複数の汎用タイマを備えていますが、C5400 プラットフォームの汎用タイマは 1 つだけです。C6000 では、構成時に CLK マネージャが使用するオンデバイス・タイマを選択できます。どのプラットフォームの場合も、タイマ割り込みがトリガされる周期を構成できます。これらのプロパティの詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「CLK Module」を参照してください。タイマ割り込みの周期を入力すると、DSP/BIOS は周期レジスタの適切な値を自動的に設定します。

C6000 プラットフォームで CLK マネージャがイネーブルの場合は、4 CPU サイクルごとにタイマ・カウンタ・レジスタが増加します。C5400 プラットフォームで CLK マネージャがイネーブルにされている場合は、タイマ・カウンタは次に示す割合で減少します。ここで、CLKOUT は MIPS 単位の DSP クロック速度で（ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「GBL」プロパティを参照）、TDDR は次の式に示すタイマ分周レジスタの値です。

```
CLKOUT / (TDDR + 1)
```

このレジスタが、C5400 および C2800 プラットフォームで 0 に達するか、C6000 プラットフォームで周期レジスタにセットされている値に達すると、カウンタはリセットされます。C5400 および C2800 では、カウンタは周期レジスタに入っている値にリセットされます。C6000 では 0 にリセットされます。この時点で、タイマ割り込みが発生します。タイマ割り込みが発生すると、選択されたタイマ用の HWI オブジェクトは CLK_F_isr 関数を実行し、その結果次のイベントが生じます。

- ❏ C6000、C2800、および C5000 プラットフォームでは、低分解能時間が 1 増加します。
- ❏ CLK オブジェクトが指定しているすべての関数が、その ISR のコンテキスト内の順序で実行されます。

したがって低分解能クロックはタイマ割り込みの頻度で進み、クロックの時間は発生したタイマ割り込みの数と同じになります。低分解能時間を取得するには、アプリケーション・コードから CLK_getltime を呼び出します。

タイマ割り込み発生時に実行される CLK 関数は、システム・クロックのティックを発生させたハードウェア割り込みのコンテキストの中で実行されます。したがって、CLK 関数の中で行われる処理量は最小限に抑えられます。そして、これらの関数は HWI の中で使用できる DSP/BIOS 呼び出しだけを起動します。

注：

HWI_enter および HWI_exit は DSP/BIOS が CLK_F_isr を実行するときに内部で呼び出されるものなので、CLK 関数はこれらの関数を呼び出してはいけません。さらに、CLK 関数では、*interrupt* キーワード、または C 関数の INTERRUPT プラグマは使用**しない**でください。

高分解能クロックは、C6000 プラットフォームではタイマ・カウンタ・レジスタが増加するのと同じレートで、また C5400 および C2800 プラットフォームではそのレジスタが減少するのと同じレートで進みます。したがって、高分解能時間はタイマ・カウンタ・レジスタが増加または減少した回数です。C6000 プラットフォームでは、これは命令サイクル数を 4 で割った値と同じです。したがって、CPU のクロック・レートが大きいと、タイマ・カウンタ・レジスタが、短時間で周期レジスタ値（C6000 プラットフォームの場合）または 0（C5400 プラットフォームの場合）に達することがあります。

C6000 プラットフォームでは、32 ビット高分解能時間を計算するには、低分解能時間（つまり、割り込みカウント）に周期レジスタの値を掛け、それにタイマ・カウンタ・レジスタの現在値を加算します。高分解能時間の値を取得するには、アプリケーション・コードから CLK_gethtime を呼び出します。最大の 32 ビット値に達すると、どちらのクロックの値も 0 からリスタートします。

C54x プラットフォームでは、32 ビット高分解能時間を計算するには、低分解能時間（つまり、割り込みカウント）に周期レジスタの値を掛け、それに、周期レジスタの値とタイマ・カウンタ・レジスタの値の差を加算します。高分解能時間の値を取得するには、アプリケーション・コードから CLK_gethtime を呼び出します。クロックの値が 0 に達すると、周期レジスタに入っている値からリスタートします。

C28x プラットフォームでは、32 ビット高分解能時間を計算するには、低分解能時間（つまり、割り込みカウント）に周期レジスタの値を掛け、それに、周期レジスタの値とタイマ・カウンタ・レジスタの値の差を加算します。高分解能時間の値を取得するには、アプリケーション・コードから CLK_gethtime を呼び出します。クロックの値が 0 に達すると、周期レジスタに入っている値からリスタートします。

その他の CLK モジュール API として、構成時に周期レジスタにセットされた値を返す CLK_getprd と、タイマ・カウンタ・レジスタの増加または減少をミリ秒単位で返す CLK_countspms があります。

低分解能クロックを構成するには、CLK マネージャのプロパティを変更します。たとえば、低分解能クロックが 1 ミリ秒（.001 秒）に 1 回進むようにするには、CLK マネージャの「Microseconds/Int」フィールドに 1000 と入力します。構成時に、周期レジスタの正しい値が自動的に計算されます。

「Directly configure on-device timer registers」プロパティを真にセットすると、周期レジスタ値を直接指定できます。C6000 プラットフォームの場合、CPU クロック /4 を使用してクロックを駆動する 160 MIPS プロセッサで 1 ミリ秒（.001 秒）のシステム・クロック周期を生成するには、周期レジスタの値を次のように指定します。

```
Period = 0.001 sec * 160,000,000 cycles per second / 4 cycles = 40,000
```

C5400 および C2800 プラットフォームの場合に、CPU を使用してクロックを駆動する 40 MIPS プロセッサで同じことをするには、周期レジスタの値を次のように指定します。

```
Period = 0.001 sec * 40,000,000 cycles per second = 40,000
```

4.9.2 システム・クロック

多くの DSP/BIOS 関数には、timeout パラメータがあります。DSP/BIOS は、システム・クロックを使用して timeout 時間が満了する時期を判断します。システム・クロックのティック速度は、低分解能時間または外部ソースにより決定することができます。

timeout パラメータをもつ関数の一例として、TSK_sleep 関数があります。この関数を呼び出した後にシステム・クロックで timeout 値と同じだけのティック数が経過すると、タイムアウトが生じます。たとえば、システム・クロックの分解能が 1 マイクロ秒のときに現行タスクが 1 ミリ秒間ブロックされるようにするには、次のように指定します。

```
/* block for 1000 ticks * 1 microsecond = 1 msec */
TSK_sleep(1000)
```

注：

プログラムの構成に PRD モジュールを駆動する機能が何も含まれていない場合は、TSK_sleep または SEM_pend には、0 または SYS_FOREVER 以外の timeout 値を指定して呼び出さないでください。デフォルト構成では、PRD モジュールを駆動するのは CLK モジュールです。

デフォルトの CLK 構成を使用している場合は PRD_clock CLK オブジェクトがシステム・クロックを駆動するので、システム・クロックの値は低分解能時間と同じです。

システム・クロックのソースとしてオンデバイス・タイマを使用することは、必須条件ではありません。代わりに、データ・ストリーム速度により駆動されるような外部クロックを使用することもできます。オンデバイス・タイマを使用して低分解能時間を駆動したくない場合は、構成スクリプトの PRD_clock という名前の CLK オブジェクトを削除します。外部クロックを使用している場合には、その外部クロックは、PRD_tick を呼び出してシステム・クロックを進めます。もう 1 つの方法として、コーデックなどのようなオンデバイス・ペリフェラルを使用して定期的に割り込みをトリガし、その割り込みの HWI から PRD_tick を呼び出すこともできます。この場合、システム・コールの分解能は、PRD_tick を呼び出している割り込みの頻度と等しくなります。

4.9.3 例 — システム・クロック

例 4-17 の clktest.c は、システム・クロックを使用する DSP/BIOS 関数 TSK_time および TSK_sleep のわかりやすい使用例を示しています。clktest.c の中の task というラベルの付いたタスクは、1000 ティックの間スリープした（ブロックされた）後で、タスク・スケジューラによりウェイクアップ（アクティブに）されます。他に作成されているタスクはないので、task がブロックされている間、プログラムはアイドル関数を実行します。このプログラムでは、システム・クロックが構成されていて、PRD_clock により駆動されることが前提になります。このプログラムは、c:\ti\examples*target*\bios\clktest フォルダに収められています（target は、ご使用のプラットフォームを表します）。例 4-17 に示すコードのトレース・ログ出力は、図 4-16 のようになります。

例 4-17. システム・クロックを使用したタスクの駆動

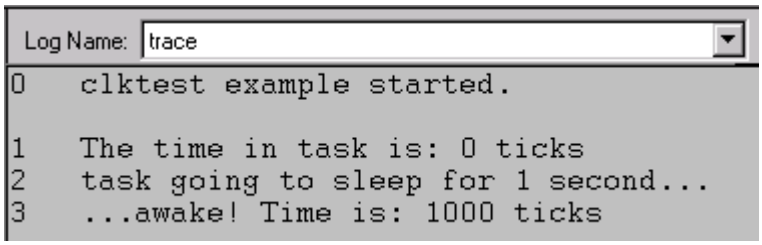
```
/*  ======== clktest.c =======
 *  In this example, a task goes to sleep for 1 sec and
 *  prints the time after it wakes up. */

#include <std.h>

#include <log.h>
#include <clk.h>
#include <tsk.h>

extern LOG_Obj trace;

/*  ======== main ======== */
Void main()
{
    LOG_printf(&trace, "clktest example started.\n");
}

Void taskFxn()
{
    Uns ticks;

    LOG_printf(&trace, "The time in task is: %d ticks",
(Int)TSK_time());

    ticks = (1000 * CLK_countspms()) / CLK_getprd();

   LOG_printf(&trace, "task going to sleep for 1 second... ");
    TSK_sleep(ticks);
    LOG_printf(&trace, "...awake! Time is: %d ticks",
(Int)TSK_time());
}
```

注：

LOG_printf に渡す非ポインタ型の関数引数は、次のコード例に示すように明示的に（Arg）に型キャストする必要があります。

```
LOG_printf(&trace, "Task %d Done", (Arg)id);
```

図 4-16.　例 4-17 のトレース・ログ出力



4.10 周期関数マネージャ（PRD）とシステム・クロック

多くのアプリケーションでは、入出力（I/O）の可用性またはその他のプログラムされたイベントに基づいて関数をスケジュールする必要があります。リアルタイム・クロックに基づいて関数をスケジュールできるアプリケーションもあります。

PRD マネージャを使用すると、プログラム関数の周期的実行をスケジュールするオブジェクトを作成できます。DSP/BIOS は、PRD モジュールを駆動するためにシステム・クロックを提供します。システム・クロックは、PRD_tick が呼び出されるたびにティックを刻む 32 ビット・カウンタです。タイマ割り込みや他の周期イベントを使用して PRD_tick を呼び出し、システム・クロックを駆動することができます。

PRD オブジェクトは複数個あっても構いませんが、すべて同じシステム・クロックにより駆動されます。PRD オブジェクトの周期によって、それに対応する関数が呼び出される頻度が決まります。PRD オブジェクトの周期は、システム・クロック時間、つまりシステム・クロック・ティック数で指定します。

特定のイベントに基づいて関数をスケジュールする手順は、次のとおりです。

- **リアルタイム・クロックをベースとする方法**：「PRD モジュール」プロパティの「Use CLK Manager to Drive PRD」プロパティを真にセットします。これで、システム・クロックを駆動するために CLK マネージャが使用するタイマ割り込みがセットされます。これにより、PRD_clock という名前の CLK オブジェクトが CLK モジュールに追加されます。このオブジェクトはタイマ割り込みがオフになるたびに PRD_tick を呼び出し、システム・クロックを 1 ティック進めます。

注：

CLK マネージャにより PRD が駆動されていると、PRD 関数を駆動するシステム・クロックは低分解能クロックと同じ速度で進みます。低分解能とシステム時刻は同調します。

- **入出力（I/O）の可用性またはその他のイベントをベースとする方法**：「PRD モジュール」プロパティの「Use the CLK Manager to Drive PRD」プロパティを偽にセットします。ユーザ・プログラムでは、システム・クロックを進ませるために PRD_tick を呼び出す必要があります。この場合、システム・クロックの分解能は、PRD_tick が呼び出される割り込みの頻度と等しくなります。

4.10.1 PRD オブジェクト用の関数を起動する方法

PRD_tick が呼び出されると、次の 2 つのことが起こります。

- PRD_D_tick（システム・クロック・カウンタ）が 1 だけ増加します。つまりシステム・クロックが進みます。

- 経過した PRD_tick の数が、PRD 関数周期の公約数の中で最大の 2 の累乗に等しくなった場合は、PRD_swi という SWI がポストされます。たとえば 3 つの PRD オブジェクトの周期が 12、24、および 36 である場合は、PRD_swi は 4 ティックごとに実行されます。12 と 6 もこれら 3 つの周期の公約数ですが、2 の累乗ではないので、12 ティックまたは 6 ティックごとに実行されることはありません。

PRD オブジェクトが静的に作成されると、PRD_swi という名前の新しい SWI オブジェクトが自動的に追加されます。

PRD_swi が実行されると、それに対応する関数は次の型のループを実行します。

```
for ("Loop through period objects") {
   if ("time for a periodic function")
       "run that periodic function";
}
```

したがって、周期関数は PRD_tick が呼び出された HWI のコンテキスト内で実行されるのではなく、PRD_swi のコンテキストまで実行を延期されます。その結果、システム・ティックが発生してからティックにより期間が経過した周期オブジェクトが実行されるまでに、遅延が生じることがあります。これらの関数をティックの直後に実行させるには、PRD_swi がアプリケーション内の他のスレッドより優先順位が高くなるように構成する必要があります。

4.10.2 PRD 統計情報と SWI 統計情報を解釈する方法

リアルタイム・システムで多くのタスクは周期的に実行されます。つまり、一定の間隔で連続的に実行されます。このようなタスクは、再度実行される時期が来るまでに実行が完了していることが重要です。この時点までに完了しないと、リアルタイム・デッドラインに違反したことになります。偶発的なデッドライン違反から回復するために内部データ・バッファリングを使用できますが、デッドライン違反が繰り返し発生すると、最終的には回復不能な障害に至る場合があります。

SWI 関数について収集される暗黙統計には、ソフトウェア割り込みが実行可能な状態になってから完了するまでの時間が示されます。プロセッサは、実際には多数のハードウェア割り込みとソフトウェア割り込みを実行するので、この時間測定は重要です。あるソフトウェア割り込みが実行可能な状態になってから他のソフトウェア割り込みが完了するまで待つ時間が長すぎると、リアルタイム・デッドライン違反が生じることになります。さらに、タスクの実行が開始された後で長時間の割り込みが何度か発生した場合も、リアルタイム・デッドライン違反が生じます。

最大所要（ready-to-complete）時間は、システム障害が発生する可能性を示す優れた判定基準になります。ソフトウェア割り込みの最大所要時間が割り込みの周期に近いほど、システムが突発的な動作の大量発生やデータ依存型の計算要件の一時的増加から回復できる可能性が低くなります。最大所要時間は、将来の製品の高機能化（これにより必要な MIPS が確実に増加する）のための余力を判断するための指針でもあります。

注：

DSP/BIOS は、各ソフトウェア割り込みの実行に要した時間を暗黙的に測定するのではありません。この測定値を計算するために、シミュレータまたはエミュレータを使用してソフトウェア割り込みが独立して実行され、必要な実行サイクルの正確な数がカウントされます。

システム内のすべてのルーチンに必要な MIPS 数の合計が DSP の MIPS 評価値より十分に小さい場合であっても、システムはリアルタイム・デッドラインの条件を満たさないことがあるという点は重要です。CPU の負荷が 70% に満たないシステムでも、優先順位付けの問題が原因でリアルタイム・デッドライン違反が生じることがよくあります。ただし、このような状況が生じた場合は、すぐに DSP/BIOS が監視する最大所要時間にその徴候が現れます。

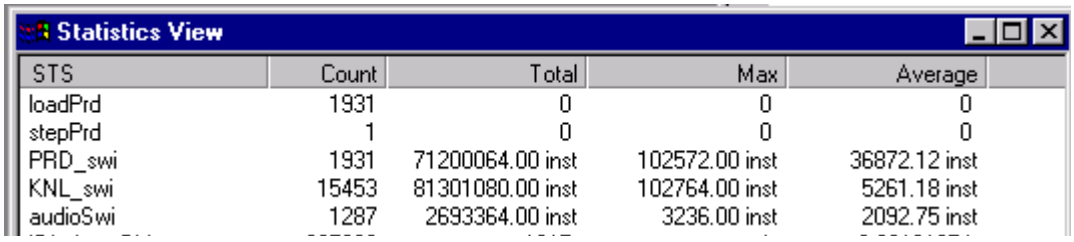
ソフトウェア割り込みおよび周期オブジェクトに関する統計アキュムレータがイネーブルにされていると、ホストは次のタイプの統計情報について、カウント、合計、最大、平均を自動的に収集します。

- **SWI**： ソフトウェア割り込みがポストされてから完了するまでに経過した時間に関する統計情報。
- **PRD**： 周期関数がレディ状態になってからその実行が完了するまでに経過した周期システム・ティックの数。定義により、周期関数がレディ状態になったと見なされるのは、period に相当するティックが発生したときです。この場合の period は、このオブジェクトの *period* パラメータです。

SWI 完了期間の単位をセットするには、CLK マネージャのパラメータをセットします。CLK モジュールの「Use high resolution time for internal timings」パラメータが「True」（デフォルト）にセットされている場合は、この期間は命令サイクル数で測定されます。この CLK パラメータが「False」にセットされている場合は、SWI 統計情報はタイマ割り込み期間の単位で表示されます。「Statistics View Property」ページの「Statistics Units」に、「ミリ秒」または「マイクロ秒」を選択することもできます。

たとえば、PRD オブジェクトの最大値が継続的に増加する場合、そのオブジェクトはおそらくリアルタイム・デッドラインの条件を満たしていません。実際、PRD オブジェクトの最大値は、この PRD オブジェクトの「period (in system ticks)」プロパティの値以下である必要があります。最大値が「period」より大きい場合（図 4-17）、周期関数はリアルタイム・デッドラインに違反しています。

図 4-17. PRD オブジェクトに関する統計ビューの使用方法



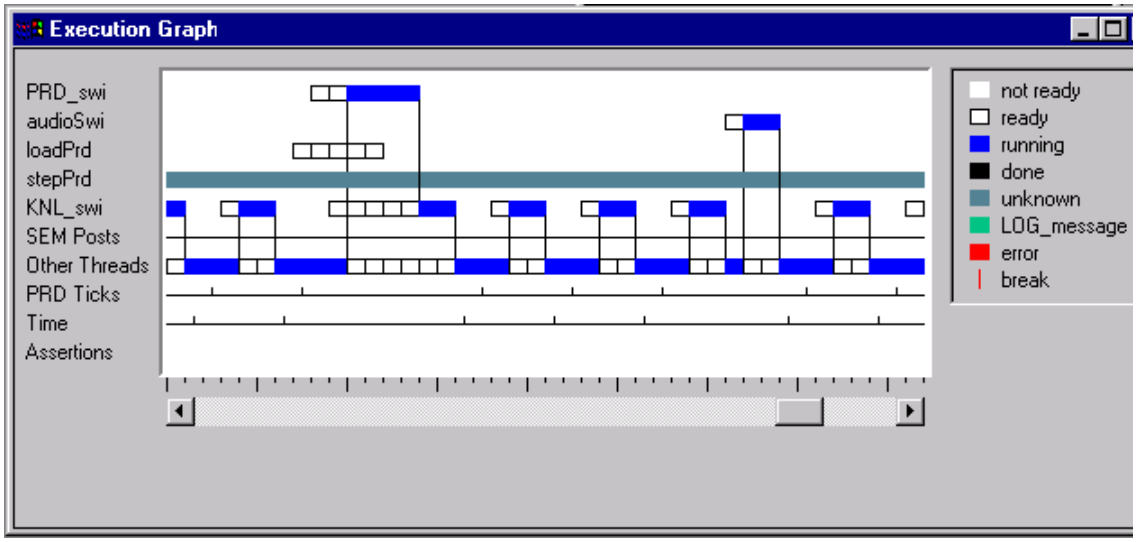
4.11 「Execution Graph」を使用してプログラムの実行状況を示す方法

Code Composer Studio で「Execution Graph」を使用して「DSP/BIOS」→「Execution Graph」の順に選択すると、スレッドの動作をビジュアルな形式で表示できます。

4.11.1 「Execution Graph」ウィンドウに表示される状態

「Execution Graph」（図 4-18）には、システム・ログ（LOG_system）の情報に基づき、タイマ割り込み（Time）およびシステム・クロック・ティック数（PRD Ticks）を基準としたスレッドの状態が表示されます。

図 4-18. 「Execution Graph」ウィンドウ



白のボックスは、スレッドがポストされ実行可能な状態になっていることを示します。青緑のボックスは、ログのこの時点では、ホストがまだこのスレッドの状態に関する情報を受け取っていないことを示します。濃青のボックスは、スレッドが実行中であることを示します。

「Errors」の行の淡青のボックスは、エラーが発生したことを示します。たとえば、「Execution Graph」にエラーが表示されるのは、スレッドがリアルタイム・デッドラインの条件を満たしていないことが検出された場合です。このボックスは、プログラムがシステム・ログを上書きしたことに起因する無効なログ・レコードも示します。エラーをダブルクリックすると、詳細な情報が表示されます。

4.11.2 「Execution Graph」ウィンドウに表示されるスレッド

左側の列に表示される SWI 関数と PRD 関数は、優先順位の高いものから順にリストされます。ただしパフォーマンス上の理由により、「Execution Graph」には、ハードウェア割り込みやバックグラウンド・スレッドに関する情報は表示されません（通常、割り込みにより行われる CLK ティックは表示されます）。SWI、PRD、または TSK スレッドが使用していないい時間は HWI または IDL スレッドに属するものなので、この時間は「OtherThreads」の行に示されます。

PIP が実行する関数（通知関数）は、PIP API を呼び出したスレッドの一部として実行されます。LNK_dataPump オブジェクトは、ホストの HST（ホスト・チャネル・マネージャ）オブジェクトの終わりを管理する関数を実行します。このオブジェクトおよび IDL バックグラウンド・スレッドから実行される他の IDL オブジェクトは、「OtherThreads」の行に含まれます。

注 :

「Time」マークと「PRD Ticks」の間隔は均等ではありません。このグラフには、イベントごとに 1 つずつ四角が表示されます。2 つの「Time」割り込みまたは「PRD Ticks」の間に多数のイベントが発生した場合、マーク間の距離は発生イベント数が少ないときの間隔より広くなります。

4.11.3 「Execution Graph」ウィンドウに表示されるシーケンス番号

図 4-18 の下部のスクロール・バーの上にある目盛りは、「Execution Graph」のイベントの順序を示します。

注 :

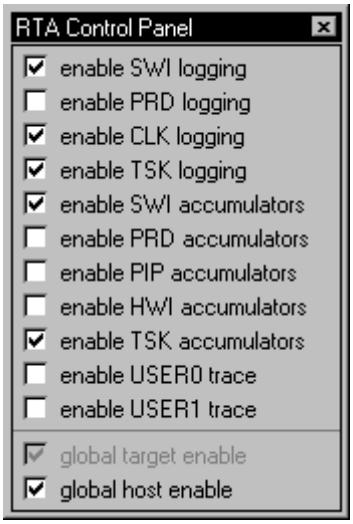
循環ログ（「Execution Graph」のデフォルト）には、最も新しい n 個のイベントだけが含まれています。通常、ホストがログをポーリングしてから次にポーリングするまでに上書きされたために、一部のイベントがログに表示されないことがあります。「Execution Graph」には、ポーリングしたログ・イベントの各グループの末尾にあるログ・シーケンス番号の中に赤の縦線とブレークが表示されます。

ログのサイズを大きくして表示されるログ・イベントの数を多くすれば、検討を要するすべてのイベントを見ることができます。また、「RTA Control Panel」を、検討したいイベントのみを記録するようにセットすることもできます。

4.11.4 「Execution Graph」を使用するために「RTA Control Panel」をセットする方法

TRC モジュールを使用すると、アプリケーション実行中の特定の時点で、「Execution Graph」に記録するイベントを制御できます。「Execution Graph」への SWI、PRD、および CLK イベントの記録は、ホストで（図 4-19 に示すように「RTA Control Panel」を使用するか、Code Composer Studio で「DSP/BIOS」→「RTA Control Panel」の順に選択して）、または（TRC_enable および TRC_disable API を介して）ターゲット・コードで制御できます。暗黙計測の制御方法の詳細については、3.3.4.2 項「暗黙計測の制御」（3-16 ページ）を参照してください。

図 4-19. 「RTA Control Panel」ダイアログ・ボックス



「Execution Graph」を使用するときに自動ポーリングをオフにしておけば、ログが頻繁にスクロールするのを防ぎ、グラフを丹念に検討するための時間をとることができます。自動ポーリングをオフにする方法は、次のとおりです。

- 「Execution Graph」を右クリックし、ショートカット・メニューから「Pause」を選択します。
- 「 RTA Control Panel」を右クリックし、「Property Page」を選択します。「Event Log/Execution Graph refresh rate」を 0 にセットしてから、「OK」をクリックします。

グラフを更新したいときは、「Execution Graph」を右クリックし、ショートカット・メニューから「Refresh Window」を選択することにより、ターゲットからログ・データをポーリングすることができます。「Refresh Window」を何回か選択すれば、追加したデータを見ることができます。グラフを右クリックしたときに表示されるショートカット・メニューを使用して、グラフに表示されている前のデータをクリアすることもできます。

実行シーケンスが複雑なプログラムで「Execution Graph」の使用を予定している場合は、構成時に LOG オブジェクト LOG_system の「buflen」プロパティの値を大きくすることで「Execution Graph」のサイズを大きくすることができます。各ログ・メッセージに 4 ワードが使用されるので、buflen は、少なくとも保存したいイベントの数に 4 を掛けた値以上にする必要があります。

C55x プラットフォームの場合は、ラージ・メモリ・モデルのデータ・ポインタの長さは 23 ビットで、ロング・ワード・アクセスには常に偶数アドレスのアライメントが必要です。したがって、ログ・イベント・バッファ・サイズは 2 倍（8 ワード）になります。

第 5 章

メモリと低レベル関数

本章では、DSP/BIOS リアルタイム・マルチタスク・カーネルに含まれている低レベル関数について説明します。低レベル関数の構成は、次のとおりです。

- ❑ MEM と BUF：可変長と固定長のメモリの割り当てを管理します。
- ❑ SYS：その他のシステム・サービスを提供します。
- ❑ QUE：キューを管理します。

本章では、これらのモジュールを使用したわかりやすいプログラム例もいくつか示します。API 関数の詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』を参照してください。

項目 **ページ**

5.1 メモリ管理

メモリ・セクション・マネージャ（MEM モジュール）は、メモリの物理範囲に対応する名前付きメモリ・セグメントを管理します。メモリ・セグメントに対する制御をさらに強化したい場合は、独自にリンカ・コマンド・ファイルを作成し、構成スクリプト実行時に作成されたリンカ・コマンド・ファイルをインクルードできます。

MEM モジュールは、可変サイズのメモリ・ブロックを動的に割り当てたり解放したりする一組の関数も提供します。BUF モジュールは、固定サイズのメモリ・ブロックを動的に割り当てたり解放したりする一組の関数を提供します。

標準の C 関数（malloc など）と異なり、MEM 関数を使用すると、特定のメモリ領域要求を満たすためにメモリのどのセグメントを使用するかを指定できます。リアルタイム DSP ハードウェア・プラットフォームには、一般にさまざまなタイプのメモリが含まれています。高速オンデバイス RAM、0 ウェイト・ステートの外部 SRAM、バルク・データ用の低速 DRAM などです。特定ブロックのデータをどのメモリ・セグメントに含めるか明示的に制御できることは、多くの DSP アプリケーションでのリアルタイムの必須条件を満たすために不可欠の要件です。

MEM モジュールは、DSP メモリ・サブシステムに関連したハードウェア・レジスタのセットまたは構成は行いません。このような構成はユーザが行う必要があり、一般に、ソフトウェア・ロード・プログラムか、Code Composer Studio の場合は GEL 起動オプションまたはメニュー・オプションから必要な設定を行います。たとえば、C6000 プラットフォームで外部メモリにアクセスするには、まず先に外部メモリ・インターフェイス（EMIF）レジスタを適切に配置してからでないと、アクセスはできません。DSP/BIOS 内部で EMIF を初期化できる最初の機会は、ユーザ初期化フックを処理する場合です（ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「Global Settings」を参照）。

MEM 関数は、可変サイズのメモリ・ブロックを割り当てと解放を行います。MEM モジュールは個々のメモリ・セグメントごとにフリー・ブロックのリンク・リストを保持しているので、このモジュールを使用した場合のメモリの割り当てや解放は非決定論方式で行われます。MEM_alloc および MEM_free は、メモリの割り当てや解放を行うときに、このリンク・リストを参照する必要があります。

5.1.1 メモリ・セグメントを構成する方法

DSP/BIOS には、メモリ・セグメントのセットを定義するいくつかのテンプレートが用意されています。これらのセグメントは、サポートされる DSP ボードごとに少しずつ異なります。構成テンプレートのないハードウェア・プラットフォームを使用する場合は、MEM オブジェクト、およびそのプロパティをカスタマイズする必要があります。MEM セグメントは、次の方法でカスタマイズできます。

- ❑ 新しい MEM セグメントを挿入し、そのプロパティを定義します。MEM オブジェクトのプロパティの詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』を参照してください。

- ❏ 既存の MEM セグメントのプロパティを変更します。。
- ❏ 一部の MEM セグメント、特に外部メモリ位置に対応するセグメントを削除します。ただし、他のオブジェクトおよびマネージャのプロパティの中で該当セグメントが参照されている場合は、まずその参照を変更しなければなりません。特定の MEM セグメントに対する依存関係を調べるには、そのセグメントで右クリックし、ポップアップ・メニューから「Show Dependencies」を選択します。IPRAM および IDRAM セグメント（C6000 プラットフォームの場合）、または IPROG および IDATA セグメント（C5000 プラットフォームの場合）は、削除もリネームも行わないことをお勧めします。
- ❏ いくつかの MEM セグメントをリネームします。セグメントをリネームするための手順は、次のとおりです。

  a) リネームするセグメントに対する依存関係を除去します。特定の MEM セグメントに対する依存関係を調べるには、そのセグメントで右クリックし、ポップアップ・メニューから「Show Dependencies」を選択します。

  b) セグメントをリネームします。セグメント名を右クリックし、ポップアップ・メニューから「Rename」を選択して名前を変更します。

  c) 必要に応じて、他のオブジェクト用のプロパティで新しいセグメント名を選択して、このセグメントに対する依存関係を再作成します。

5.1.2 動的メモリ割り当てをディセーブルにする方法

コード・サイズが小さいことがアプリケーションにとって重要な要件である場合は、メモリを動的に割り当てたり解放したりする機能を除去することにより、コード・サイズを大幅に削減できます。この機能を除去した場合、ユーザ・プログラムは、MEM 関数またはオブジェクト作成関数（TSK_create など）を呼び出すことはできません。したがって、プログラムで使用するすべてのオブジェクトを構成時に作成する必要があります。

動的メモリ割り当て機能を解除するには、MEM マネージャの「No Dynamic Memory Heaps」を真にセットします。

動的メモリ割り当てがディセーブルにされているときにプログラムが MEM 関数を呼び出すと（またはオブジェクト作成関数を呼び出すことにより間接的に MEM 関数を呼び出すと）、エラーが発生します。MEM 関数に渡された segid がセグメントの名前である場合は、リンク・エラーが発生します。MEM 関数に渡された segid が整数である場合、MEM 関数は SYS_error を呼び出します。

5.1.3 ユーザ独自のリンカ・コマンド・ファイル内でセグメントを定義する方法

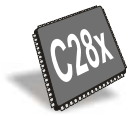
MEM マネージャを使用すると、各種のコード型およびデータをどのメモリ・セグメントに入れるかを選択することができます。コードやデータをどこに格納するかについての制御を強化するには、MEM マネージャの「User .cmd file for non-DSP/BIOS segments」プロパティを真にセットします。

次に、独自のリンカ・コマンド・ファイルを作成し、その始めに構成スクリプトを実行して作成されたリンカ・コマンド・ファイルを組み込みます。たとえば、ユーザ独自のリンカ・コマンド・ファイルは例 5-1 または例 5-2 のようになります。

例 5-1. リンカ・コマンド・ファイル（C6000 プラットフォーム）


```
/* First include DSP/BIOS generated cmd file. */
-l designcfg.cmd

SECTIONS {
   /* place high-performance code in on-device ram */
   .fast_text: {
      myfastcode.lib*(.text)
      myfastcode.lib*(.switch)
   } > IPRAM

   /* all other user code in off device ram */
      .text:    {} > SDRAM0
      .switch:  {} > SDRAM0
      .cinit:   {} > SDRAM0
      .pinit:   {} > SDRAM0

   /* user data in on-device ram */
      .bss:     {} > IDRAM
      .far:     {} > IDRAM
}
```

例 5-2. リンカ・コマンド・ファイル（C5000 および C28x プラットフォーム）




```
/* First include DSP/BIOS generated cmd file. */
-l designcfg.cmd

SECTIONS {
   /* place high-performance code in on-device ram */
   .fast_text: {
      myfastcode.lib*(.text)
      myfastcode.lib*(.switch)
   } > IPROG PAGE 0

   /* all other user code in off device ram */
      .text:    {} > EPROG0 PAGE 0
      .switch:  {} > EPROG0 PAGE 0
      .cinit:   {} > EPROG0 PAGE 0
      .pinit:   {} > EPROG0 PAGE 0

   /* user data in on-device ram */
      .bss:     {} > IDATA PAGE 1
      .far:     {} > IDATA PAGE 1
}
```

5.1.4 メモリを動的に割り当てる方法

DSP/BIOS は､動的なメモリ割り当てを行うために 2 つのモジュール（MEM と BUF）で複数の関数を提供します。MEM モジュールは可変サイズのメモリ・ブロックを割り当てます。BUF モジュールはバッファ・プールから固定サイズのバッファを割り当てます。

5.1.4.1 MEM モジュールを使用したメモリ割り当て

基本的な記憶域割り当ては、MEM_alloc により行われます。この関数のパラメータは、例 5-3 に示すように、メモリ・セグメント、ブロック・サイズ、アライメントを指定します。メモリ要求を満たすことができない場合、MEM_alloc は MEM_ILLEGAL を返します。

例 5-3. システム・レベルの記憶域に MEM_alloc を使用する方法

```
Ptr MEM_alloc(segid, size, align)
    Int segid;
    Uns size;
    Uns align;
```

segid パラメータは、どのメモリ・セグメントからメモリを割り当てるかを指定します。この識別子は、整数または構成時に定義したメモリ・セグメント名です。

MEM_alloc が返すメモリ・ブロックには、少なくとも、size パラメータに示されている最小アドレス可能データ単位（MADU）と同数が含まれます。プロセッサの場合の最小アドレス可能単位は、メモリにロードまたはストアできる最小のデータです。MADU は、連続したメモリ・アドレス間のビット数と考えることができます。MADU のビット数は DSP デバイスによって異なります。たとえば、C5000 プラットフォームの場合の MADU は 16 ビット・ワードで、C6000 プラットフォームの場合の MADU は 8 ビット・ワードです。

MEM_alloc が返すメモリ・ブロックは、align パラメータの倍数に相当するアドレスで始まります。align が 0 の場合はアライメントに関する制約はありません。構造体の配列は、例 5-4 に示すように割り当てられます。

例 5-4. 構造体の配列を割り当てる方法

```
typedef struct Obj {
    Int    field1;
    Int    field2;
    Ptr    objArr;
} Obj;

objArr = MEM_alloc(SRAM, sizeof(Obj) * ARRAYLEN, 0);
```

多くの DSP アルゴリズムでは、サーキュラ・バッファが使用されています。サーキュラ・バッファが 2 の累乗の境界にアライメントされていれば、ほとんどの DSP で効率的に管理できます。このバッファ・アライメントにより、多くの DSP で採用されているサーキュラ・アドレッシング・モードの利点をコードに活用することができます。

アライメントが不要な場合は、align には 0 を指定します。MEM の実装では、align の値が 0 であっても、メモリは MEM_Header 構造体を 1 つ収容するために必要なワード数に相当する境界でアライメントされます。align に 0 以外の値を指定した場合、メモリは align が 2 の累乗に相当するアライメント・ワード境界に割り当てられます。

MEM_free は、MEM_alloc、MEM_calloc、または MEM_valloc への以前の呼び出しにより取得したメモリを解放します。MEM_free のパラメータ（segid、ptr、size）は、例 5-5 に示すようにメモリ・セグメント、ポインタ、およびブロック・サイズをそれぞれ指定します。これらのパラメータの値は、記憶域ブロックを割り当てるときに使用した値と同じでなければなりません。

例 5-5.　MEM_free を使用してメモリを解放する方法

```
Void MEM_free(segid, ptr, size)
    Int segid;
    Ptr ptr;
    Uns size;
```

例 5-6 に、例 5-5 で割り当てられたオブジェクトの配列を解放する関数呼び出しを示します。

例 5-6.　オブジェクトの配列を解放する方法

```
MEM_free(SRAM, objArr, sizeof(Obj) * ARRAYLEN);
```

5.1.4.2 BUF モジュールを使用したメモリ割り当て

BUF モジュールは、固定サイズのバッファ・プールを保持しています。固定サイズのバッファ・プールは、静的または動的に作成することができます。動的に作成されたバッファ・プールは、MEM モジュールが管理している動的なメモリ・ヒープから割り当てられます。BUF_create 関数は、バッファ・プールを動的に作成します。設計時にサイズとアライメントに関する制限がわかっていれば、通常、アプリケーションのバッファ・プールは静的に作成します。バッファ・プールを実行時に作成するのは、前記の制限が実行時に変化する場合です。

バッファ・プール内では、すべてのバッファのサイズとアライメントは同じです。各フレームは固定長ですが、アプリケーションは、フレームの長さ以下なら各フレームに可変量のデータを入れることができます。バッファ・サイズが異なる複数のバッファ・プールを作成することができます。

BUF_alloc 関数と BUF_free 関数を使用して必要に応じて実行時に、プールからバッファを割り当てたり解放したりすることができます。

MEM モジュールが提供する動的なメモリ・ヒープからではなく、バッファ・プールからメモリを割り当てる利点は次のとおりです。

- **決定論的な割り当て時間：**BUF_alloc 関数と BUF_free 関数には、一定の時間が必要です。ヒープからのメモリ割り当てと解放は、決まっているとはいえません。
- **すべてのスレッドの型から呼び出し可能：**バッファの割り当てと解放は、極小的でかつ非ブロッキングです。したがって、BUF_alloc と BUF_free は、DSP/BIOS スレッドのすべての型（HWI、SWI、TSK、IDL）から呼び出すことができます。これに対して、HWI と SWI のスレッドは MEM_alloc を呼び出すことができません。
- **固定長の割り当ての最適化：**固定長とは対照的に、可変長の割り当てでは、MEM_alloc が最適化されます。
- **断片化の減少：**バッファが固定サイズのため、プールの断片化は発生しません。

5.1.5 メモリ・セグメントのステータスを確認する方法

MEM_stat を使用すると、メモリ・セグメントのステータスを、最小アドレス可能データ単位（MADU）の数で表示できます。MEM_alloc、MEM_calloc、および MEM_valloc（例 5-3 を参照）の場合と同様に、MEM_stat は使用サイズおよび長さの値を返します。

BUF モジュールを使用している場合、BUF_stat を呼び出してバッファ・プールの統計情報を確認することができます。また、BUF_maxbuff を呼び出して、プール内で使用しているバッファの最大数を確認することもできます。

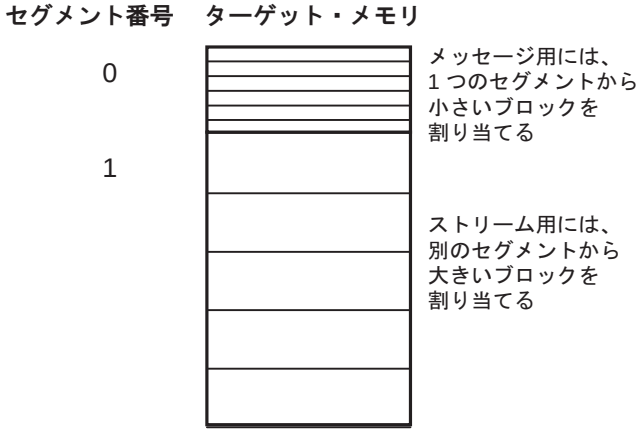
5.1.6 メモリの断片化を緩和する方法

前述のとおり、BUF モジュールを使用してバッファ・プールから固定長のバッファを割り当てたり解放したりすると、メモリの断片化が緩和されます。

可変サイズのメモリ・ブロックの割り当てと解放を何度も繰り返すと、メモリの断片化が生じることがあります。断片化により、要求を満たせるだけの大きさを備えた連続したフリー・メモリのブロックがなくなると、MEM_alloc を呼び出したときに MEM_ILLEGAL が返されることがあります。すべてのフリー・メモリ・ブロックの合計メモリ量が要求された量を上回っていても、この状況が発生します。

メモリの断片化を最小限に抑えるためには、図 5-1 に示すように、可変サイズのメモリ・ブロックを割り当てるときに、割り当てのサイズに応じて異なるメモリ・セグメントを使用することができます。

図 5-1.　異なるサイズのメモリ・セグメントの割り当て



注 :

メモリの断片化を最小限に抑えるには、1 つのメモリ・セグメントから小さい同一サイズのメモリ・ブロックを割り当て、別のメモリ・セグメントから大きい同一サイズのメモリ・ブロックを割り当てるようにします。

5.1.7　MEM の例

例 5-7 および 例 5-8 に、関数 MEM_stat、MEM_alloc、および MEM_free を使用して、メモリ割り当てに関連したいくつかの要点を示します。図 5-2 に、例 5-7 または例 5-8 の結果のトレース・ウィンドウを示します。

例 5-7 および例 5-8 では、メモリは MEM_alloc を使用して IDATA および IDRAM メモリから割り当てられ、後で MEM_free を使用して解放されます。printmem を使用して、メモリ・ステータスがトレース・バッファに出力されます。最終値（たとえば「after freeing...」）は、初期値に一致しなければなりません。

例 5-7.　メモリ割り当て（C5000 および C28x プラットフォーム）


```
/*  ======== memtest.c ========
 *  This code allocates/frees memory from different memory segments.
 */

#include <std.h>
#include <log.h>
#include <mem.h>

#define NALLOCS 2       /* # of allocations from each segment */
#define BUFSIZE 128     /* size of allocations */

/* "trace" Log created by Configuration Tool */
extern LOG_Obj trace;
#ifdef -54-
extern Int IDATA;
#endif
#ifdef -55-
extern Int DATA;
#endif
static Void printmem(Int segid);
/*
 *  ======== main ========
 */
Void main()
{
    Int i;
    Ptr ram[NALLOCS];
    LOG_printf(&trace, "before allocating ...");
    /* print initial memory status */
    printmem(IDATA);
    LOG_printf(&trace, "allocating ...");
    /* allocate some memory from each segment */
    for (i = 0; i < NALLOCS; i++) {
        ram[i] = MEM_alloc(IDATA, BUFSIZE, 0);
        LOG_printf(&trace, "seg %d: ptr = 0x%x", IDATA, ram[i]);
    }
    LOG_printf(&trace, "after allocating ...");
    /* print memory status */
    printmem(IDATA);
    /* free memory */
    for (i = 0; i < NALLOCS; i++) {
        MEM_free(IDATA, ram[i], BUFSIZE);
    }
    LOG_printf(&trace, "after freeing ...");
    /* print memory status */
    printmem(IDATA);
}
/*
 *  ======== printmem ========
 */
static Void printmem(Int segid)
{
    MEM_Stat    statbuf;
    MEM_stat(segid, &statbuf);
    LOG_printf(&trace, "seg %d: O 0x%x", segid, statbuf.size);
    LOG_printf(&trace, "\tU 0x%x\tA 0x%x", statbuf.used, stat-
buf.length);
}
```

注：

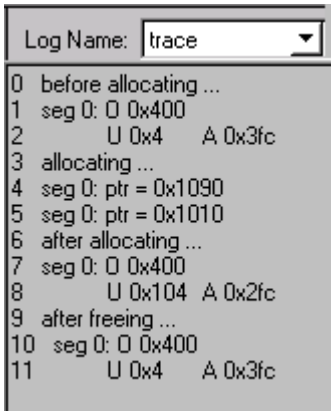
LOG_printf に渡す非ポインタ型の関数引数は、次のコード例に示すように明示的に（Arg）に型キャストする必要があります。

```
LOG_printf(&trace, "Task %d Done", (Arg)id);
```

例 5-8.　メモリ割り当て（C6000 プラットフォーム）


```
/*  ======== memtest.c ========
 *  This program allocates and frees memory from
 *  different memory segments.
 */

#include <std.h>
#include <log.h>
#include <mem.h>

#define NALLOCS 2       /* # of allocations from each segment */
#define BUFSIZE 128     /* size of allocations */

/* "trace" Log created by Configuration Tool */
extern LOG_Obj trace;
extern Int IDRAM;
static Void printmem(Int segid);

/*
 *  ======== main ========
 */
Void main()
{
    Int i;
    Ptr ram[NALLOCS];
    LOG_printf(&trace, "before allocating ...");
    /* print initial memory status */
    printmem(IDRAM);
    LOG_printf(&trace, "allocating ...");
    /* allocate some memory from each segment */
    for (i = 0; i < NALLOCS; i++) {
        ram[i] = MEM_alloc(IDRAM, BUFSIZE, 0);
       LOG_printf(&trace, "seg %d: ptr = 0x%x", IDRAM, ram[i]);
    }
    LOG_printf(&trace, "after allocating ...");
    /* print memory status */
    printmem(IDRAM);
    /* free memory */
    for (i = 0; i < NALLOCS; i++) {
        MEM_free(IDRAM, ram[i], BUFSIZE);
    }
    LOG_printf(&trace, "after freeing ...");
    /* print memory status */
    printmem(IDRAM);
}
/*
 *  ======== printmem ========
 */
static Void printmem(Int segid)
{
    MEM_Stat    statbuf;
    MEM_stat(segid, &statbuf);
    LOG_printf(&trace, "seg %d: O 0x%x", segid, statbuf.size);
    LOG_printf(&trace, "\tU 0x%x\tA 0x%x", statbuf.used, stat-
buf.length);
}
```

図 5-2.　メモリ割り当てのトレース・ウィンドウ



例 5-7 と例 5-8 のプログラムの結果は、ボードによって異なります。O は当初のメモリ量、U は使用されたメモリ量、A は連続したフリー・メモリの最大ブロック長（MADU の数）を示しています。実際に表示されるアドレスは、例 5-2 に示すものとは違うものになります。

5.2 システム・サービス

SYS モジュールは、標準の C ランタイム・ライブラリに通常収められている、よく似た関数を模した基本的なシステム・サービスのセットを提供します。原則として DSP/BIOS ソフトウェア・モジュールは、同等の C ライブラリ関数の代わりに SYS が提供するサービスを使用します。

プログラムが SYS 関数のいずれかを呼び出したときに必ず実行される関数を構成できます。詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の SYS に関する説明を参照してください。

5.2.1 実行を停止する方法

SYS には、例 5-9 に示すようにプログラムの実行を停止するための 2 つの関数があります。正常終了を行うための SYS_exit と、破滅的な状況が生じた場合に対処するために予約されている SYS_abort です。プログラムの終了または異常終了時に行われる動作は本質的にシステムに依存するものなので、ユーザは構成設定を変更して、SYS_exit または SYS_abort が呼び出されるたびにユーザ独自のルーチンが起動されるようにできます。

例 5-9. プログラムの実行を停止する SYS_exit と SYS_abort のコーディング

```
Void SYS_exit(status)
    Int status;

Void SYS_abort(format, [arg,] ...)
    String  format;
    Arg arg;
```

例 5-9 に示した関数は、SYS モジュールの「Exit function」プロパティと「Abort function」プロパティに指定されているルーチンを呼び出すことにより実行を終了します。デフォルトの終了関数は UTL_halt です。デフォルトの異常終了関数は _UTL_doAbort です。これはエラー・メッセージをログに記録し、_halt を呼び出します。_halt 関数は boot.c ファイルの中で定義されていて、すべてのプロセッサ割り込みをディセーブルにし無期限にループします。

SYS_abort は、書式文字列とオプションの一組のデータ値（多くの場合、診断メッセージを表すもの）を受け取り、例 5-10 に示すように、SYS モジュールの「Abort function」プロパティに指定されている関数に渡します。

例 5-10. オプションのデータ値を伴う SYS_abort を使用する方法

```
(*(Abort_function)) (format, vargs)
```

単一の vargs パラメータは va_list 型のもので、最初に SYS_abort に渡された arg パラメータのシーケンスを表します。「Abort function」プロパティに指定されている関数では、プログラムの実行終了前に、format パラメータと vargs パラメータを SYS_vprintf または SYS_vsprintf に直接渡す操作を選択することもできます。SYS_vprintf または SYS_vsprintf の大きいコードがもたらすオーバーヘッドを回避するために、代わりに LOG_error を使用して単に書式文字列を出力するだけにすることもできます。

上記と同様に、SYS_exit も「Exit function」プロパティに指定されている関数を呼び出してオリジナルの status パラメータを渡します。SYS_exit は、例 5-11 に示すように、まず SYS_atexit 関数により登録されている一連のハンドラを実行します。

例 5-11. SYS_exit でハンドラを使用する方法

```
(*handlerN)(status)
        ...
(*handler2)(status)
(*handler1)(status)

(*(Exit_function))(status)
```

SYS_atexit 関数が提供するメカニズムを使用して、例 5-12 に示すように、SYS_NUMHANDLERS（これは 8 にセットされます）までの範囲内で、クリーンアップ・ルーチンをスタックできます。ハンドラは、SYS_exit が「Exit function」プロパティに指定された関数を呼び出す前に実行されます。SYS_atexit は、内部スタックが満杯のときは FALSE を返します。

例 5-12. 複数の SYS_NUMHANDLERS を使用する方法

```
Bool SYS_atexit(handler)
    Fxn    handler;
```

5.2.2 エラーを処理する方法

DSP/BIOS のエラー条件を処理するには、SYS_error を使用します（例 5-13 を参照）。アプリケーション・プログラムも内部関数と同様に、SYS_error を使用してプログラム・エラーを処理します。

例 5-13. DSP/BIOS エラー処理

```
Void SYS_error(s, errno, ...)
    String     s;
    Uns    errno;
```

SYS_error は、「Error function」プロパティに指定されている関数を使用してエラー条件を処理します。構成テンプレート内のデフォルトのエラー関数は _UTL_doError で、これはエラー・メッセージをログに記録するものです。例 5-14 に示すように、「Error function」は、LOG_error を使用してエラー番号および関連のエラー文字列を出力する doError を使用するように設定できます。

例 5-14. doError を使用してエラー情報を印刷する方法

```
Void doError(String s, Int errno, va_list ap)
{
    LOG_error("SYS_error called: error id = 0x%x", errno);
    LOG_error("SYS_error called: string = '%s'", s);
}
```

errno に渡される SYS_error パラメータは、DSP/BIOS エラー（たとえば SYS_EALLOC）の場合と、ユーザ・エラー（errno >= 256）の場合があります。エラー・コードと文字列の表については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』を参照してください。

注：

メモリ量と CPU 要件を増やすエラー・チェック機能は、DSP/BIOS API では必要最小限に抑えられています。また、API リファレンスでは、DSP/BIOS API 関数を呼び出す際の制約事項を示しています。アプリケーション開発者は、これらの制約事項を守る責任があります。

5.3 キュー

QUE モジュールは、QUE エレメントのリストを管理するための一連の関数を提供します。リスト内のどこにでもエレメントを挿入したり削除したりできますが、QUE モジュールは通常 FIFO リストを実装するために使用されます。FIFO リストではエレメントはリストの末尾に追加され、リストの先頭から削除されます。QUE エレメントは、最初のフィールドの型が QUE_Elem であればどのような構造体でも構いません。例 5-15 では、QUE モジュールは QUE_Elem を使用して構造体をエンキューします。残りのフィールドには、エンキューする実際のデータが入っています。

キューを作成したり削除したりするには、QUE_create および QUE_delete をそれぞれ使用します。QUE キューはリンク・リストとして実装されるので、キューには最大制限サイズはありません。例 5-15 に、これを示します。

例 5-15. キューを使用して QUE エレメントを管理する方法

```
typedef struct QUE_Elem {
    struct QUE_Elem *next;
    struct QUE_Elem *prev;
} QUE_Elem;

typedef struct MsgObj {
    QUE_Elem   elem;
    Char       val;
} MsgObj;

QUE_Handle QUE_create(attrs)
    QUE_Attrs  *attrs;

Void QUE_delete(queue)
    QUE_Handle queue;
```

5.3.1 極小 QUE 関数

QUE_put および QUE_get は、キューの末尾へのエレメントの挿入、またはキューの先頭からのエレメントの削除を極小的に行うために使用します。これらは、割り込みをディセーブルにした状態でエレメントが挿入または削除されるという極小関数です。したがって、複数のスレッドが外部的な同期をまったく必要とせずに、安全にこの 2 つの関数を使用してキューを修正できます。

QUE_get は極小的にキューの先頭からエレメントを除去し、そのエレメントを返します。QUE_put は、極小的にキューの末尾にエレメントを挿入します。どちらの関数の場合も、例 5-16 に示すように、不要な型キャストを回避するためにキュー・エレメントの型は Ptr になっています。

例 5-16. キューへ極小的に挿入する方法

```
Ptr QUE_get(queue)
    QUE_Handle queue;
Ptr QUE_put(queue, elem)
    QUE_Handle queue;
    Ptr     elem;
```

5.3.2 その他の QUE 関数

QUE_get や QUE_put と異なり、割り込みをディセーブルにせずにキューを更新する QUE 関数もいくつかあります。変更しようとしているキューを複数のスレッドが共有している場合、これらの関数は何らかの相互排他メカニズムと一緒に使用する必要があります。

QUE_dequeue と QUE_enqueue は、割り込みをディセーブルにせずにキューを更新するという点を除けば、QUE_get や QUE_put と同じ働きをします。

QUE_head は、キューの最初のエレメントを指すポインタを（そのエレメントを除去せずに）返したい場合に使用します。QUE_next と QUE_prev は、キュー内のエレメントをスキャンするために使用します。QUE_next はキュー内の次のエレメントを指すポインタを返し、QUE_prev はキュー内の前のエレメントを指すポインタを返します。

QUE_insert および QUE_remove は、キュー内の任意の位置でエレメントを挿入したり削除したりするために使用します。

例 5-17. 相互排他エレメントに対して QUE 関数を使用する方法

```
Ptr QUE_dequeue(queue)
    QUE_Handle queue;

Void QUE_enqueue(queue, elem)
    QUE_Handle queue;
    Ptr     elem;

Ptr QUE_head(queue)
    QUE_Handle queue;

Ptr QUE_next(qelem)
    Ptr qelem;

Ptr QUE_prev(qelem)
    Ptr qelem;
Void QUE_insert(qelem, elem)
    Ptr qelem;
    Ptr elem;

Void QUE_remove(qelem)
    Ptr qelem;
```

注：

QUE キューはヘッダ・ノード付きのダブル・リンク・リストとして実装されるので、QUE_head、QUE_next、または QUE_prev がヘッダ・ノード自体（たとえば空キュー上の QUE_head を呼び出す）を指すポインタを返すことがあります。QUE_remove を呼び出してこのヘッダ・ノードを削除することのないように注意してください。

5.3.3 QUE の例

例 5-18 では、QUE キューを使用してライタ・タスクからリーダ・タスクに 5 つのメッセージを送信しています。関数 MEM_alloc および MEM_free は、MsgObj 構造体の割り当ておよび解放を行うために使用します。

例 5-18 のプログラムの結果は、図 5-3 のようになります。ライタ・タスクは QUE_put を使用して 5 つのメッセージをエンキューし、リーダ・タスクは QUE_get を使用して各メッセージをデキューします。

例 5-18. QUE を使用したメッセージの送信

```
/*
 *  ======== quetest.c ========
 *  Use a QUE queue to send messages from a writer to a read
 *  reader.
 *
 *  The queue is created by the Configuration Tool.
 *  For simplicity, we use MEM_alloc and MEM_free to manage
 *  the MsgObj structures. It would be way more efficient to
 *  preallocate a pool of MsgObj's and keep them on a 'free'
 *  queue. Using the Config Tool, create 'freeQueue'. Then in
 *  main, allocate the MsgObj's with MEM_alloc and add them to
 *  'freeQueue' with QUE_put.
 *  You can then replace MEM_alloc calls with QUE_get(freeQueue)
 *  and MEM_free with QUE_put(freeQueue, msg).
 *
 *  A queue can hold an arbitrary number of messages or elements.
 *  Each message must, however, be a structure with a QUE_Elem as
 *  its first field.
 */

#include <std.h>
#include <log.h>
#include <mem.h>
#include <que.h>
#include <sys.h>

#define NUMMSGS     5       /* number of messages */

typedef struct MsgObj {
    QUE_Elem    elem;       /* first field for QUE */
    Char        val;        /* message value */
} MsgObj, *Msg;

extern QUE_Obj queue;

/* Trace Log created statically */
extern LOG_Obj trace;

Void reader();
Void writer();
```

例 5-18.　QUE を使用したメッセージの送信（続き）

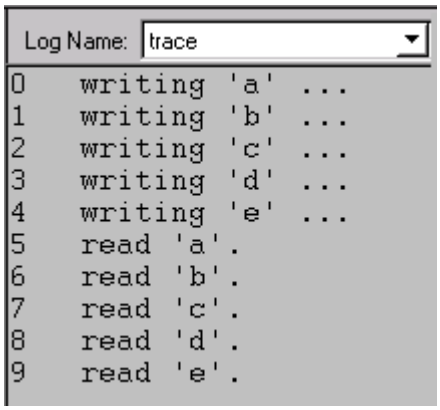
```
/*  ======== main ======== */
Void main()
{
    /*
     *  Writer must be called before reader to ensure that the
     *  queue is non-empty for the reader.
     */
    writer();
    reader();
}

/*  ======== reader ======== */
Void reader()
{
    Msg         msg;
    Int         i;
    for (i=0; i < NUMMSGS; i++) {
        /* The queue should never be empty */
        if (QUE_empty(&queue)) {
            SYS_abort("queue error\n");
        }
        /* dequeue message */
        msg = QUE_get(&queue);

        /* print value */
        LOG_printf(&trace, "read '%c'.", msg->val);
        /* free msg */
        MEM_free(0, msg, sizeof(MsgObj));
    }
}

/*  ======== writer ======== */
Void writer()
{
    Msg         msg;
    Int         i;
    for (i=0; i < NUMMSGS; i++) {
        /* allocate msg */
        msg = MEM_alloc(0, sizeof(MsgObj), 0);
        if (msg == MEM_ILLEGAL) {
            SYS_abort("Memory allocation failed!\n");
        }
        /* fill in value */
        msg->val = i + 'a';
        LOG_printf(&trace, "writing '%c' ...", msg->val);
        /* enqueue message */
        QUE_put(&queue, msg);
    }
}
```

注：

log_printf に渡す非ポインタ型の関数引数は、次のコード例に示すように明示的に（Arg）に型キャストする必要があります。

```
LOG_printf(&trace, "Task %d Done", (Arg)id);
```

図 5-3.　例 5-18 の結果を示すトレース・ウィンドウ



第 6 章

入出力（I/O）方式

本章では、DSP/BIOS データ転送方式の概要を示し、特にパイプについて説明します。

項目 **ページ**

6.1 入出力（I/O）の概要

アプリケーション・レベルでは、入出力はストリーム、パイプ、メッセージ・キュー、またはホスト・チャネルのオブジェクトにより処理されます。それぞれのオブジェクト型には、データの入出力を管理するための専用モジュールがあります。

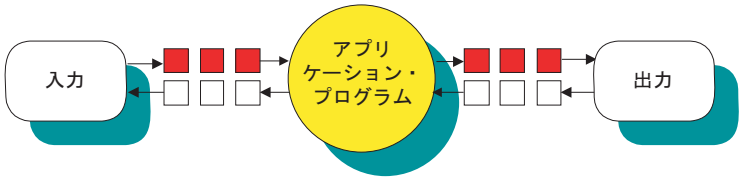
注：

パイプとストリームの代替としては、GIO クラス・ドライバを使用して IOM ミニドライバとインターフェイスします。『DSP/BIOS Driver Developer's Guide』（文献番号 SPRU616）では、GIO クラス・ドライバと IOM ミニドライバ・モデルについて解説しています。

IOM ミニドライバをストリームまたはパイプと一緒に使用している場合は、本章のストリームとパイプ・オブジェクトに関する情報が適用されます。

ストリームは、アプリケーション・プログラムと入出力（I/O）デバイスとの間でデータが流れるために通過するチャネルです。このチャネルは、読み取り専用（入力）または書き込み専用（出力）となります（図 6-1 を参照）。ストリームは、すべての入出力（I/O）デバイスに単純で汎用性のあるインターフェイスを提供し、アプリケーションが個々のデバイスの動作の詳細をまったく知る必要がないようにします。

図 6-1. 入出力（I/O）ストリーム



ストリーム入出力（I/O）の重要な性質は、その非同期性です。計算と並行して、いくつものバッファのデータの入力または出力のオペレーションが行われます。アプリケーションが現行バッファを処理している間に、新しい入力バッファへの入力オペレーション、および前のバッファからの出力オペレーションが行われます。この効率的な入出力（I/O）バッファ管理により、ストリームでのデータのコピー量が最小限に抑えられます。ストリームは、データではなくポインタを交換します。したがってオーバーヘッドが削減され、プログラムはリアルタイムの制約条件を容易に満たすことができます。

一般的なプログラムは、1 バッファ分の入力データを受け取り、そのデータを処理し、最後に 1 バッファ分の処理済みデータを出力します。そしてプログラムが終了するまで、このシーケンスは何度も繰り返されます。

D/A コンバータ、ビデオ・フレーム・グラバ、トランスデューサ、および DMA チャネルは、よく使用される入出力（I/O）デバイスのほんの一例です。ストリーム・モジュール（SIO）は、DSP/BIOS プログラミング・インターフェイスを使用する（DEV モジュールの管理下にある）デバイスを介して、これらのさまざまなタイプのデバイスと対話します。

データ・パイプは、入出力データのストリームをバッファするために使用します。これらのデータ・パイプは、DSP デバイスとすべての種類のリアルタイム・ペリフェラル・デバイスとの間の入出力（I/O）を駆動するために使用できる、一貫性のあるソフトウェア・データ構造を提供します。データ・パイプの場合は、ストリームよりオーバーヘッドが大きくなります。通知は、パイプ・マネージャにより自動処理されます。パイプでは、すべての入出力（I/O）オペレーションは一度に 1 フレームずつ処理されます。各フレームは固定長ですが、アプリケーションはフレームの長さ以下なら各フレームに可変量のデータを入れることができます。

データ転送スレッドごとに別個のパイプを使用する必要があります。また 1 つのパイプには単一のリーダと単一のライタがあり、ポイント・ツー・ポイントの通信を提供している必要があります。多くの場合、パイプの一端は HWI で制御され、もう一端は SWI 関数で制御されます。パイプを使用すると、2 つのアプリケーション・スレッド間でデータを転送することもできます。

メッセージ・キューを使用すると、可変長メッセージの構造化された送受信が可能になります。このモジュールは、複数のプロセッサ間におけるメッセージングに使用できます。メッセージ・キューについては、6.5 節「メッセージ・キュー」を参照してください。

ホスト・チャネル・オブジェクトを使用すると、アプリケーションはターゲットとホストの間でデータをストリーム方式で転送することができます。ホスト・チャネルは、入力または出力用に静的に構成されます。各ホスト・チャネルは、内部的にはデータ・パイプ・オブジェクトを使用して実現されています。

6.2 パイプとストリームの比較

DSP/BIOS は、2 つの異なるデータ転送モデルをサポートします。パイプ・モデルは、PIP モジュールと HST モジュールで使用されます。ストリーム・モデルは、SIO モジュールと DEV モジュールで使用されます。

どちらのモデルでも、パイプまたはストリームに単一のリーダ・スレッドと単一のライタ・スレッドが必要です。どちらのモデルも、データではなくポインタをバッファ間でコピーすることによって、パイプまたはストリーム内部でバッファの転送を行います。

一般に、パイプ・モデルは低レベルの通信をサポートし、ストリーム・モデルは高レベルのデバイスに依存しない入出力（I/O）をサポートします。表 6-1 では、これらの 2 種類のモデルをさらに詳しく比較します。

表 6-1. パイプとストリームの比較

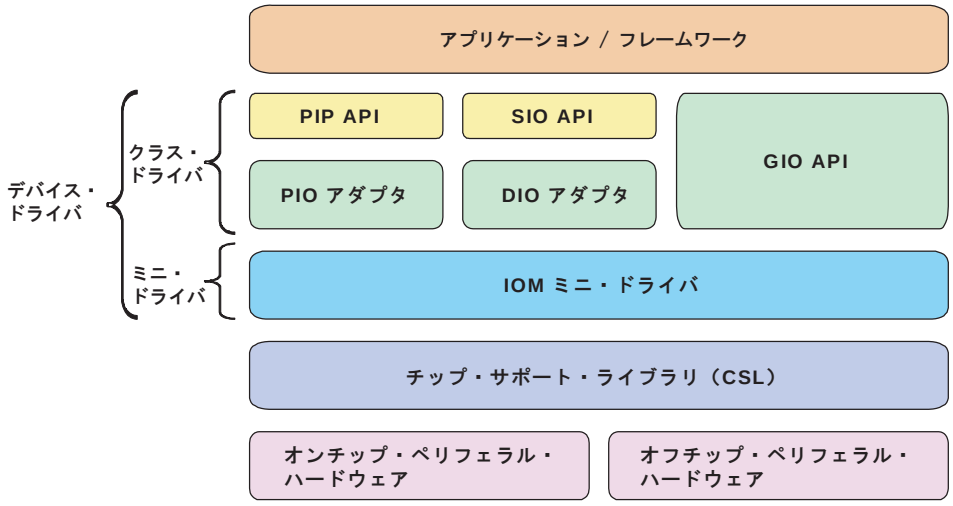
パイプ（PIP および HST）	ストリーム（SIO および DEV）
プログラマが独自のドライバ構造を作成する必要があります。	より構造化された方式で、デバイス・ドライバを作成することができます。

パイプ（PIP および HST）	ストリーム（SIO および DEV）
リーダとライタには、任意のスレッド・タイプまたはホスト PC を使用することができます。	一端は、SIO 呼び出しを使用するタスク（TSK）で処理する必要があります。もう一端は、Dxx 呼び出しを使用する HWI で処理する必要があります。
PIP 関数は非ブロッキング関数です。プログラムは、パイプからの読み取りまたはパイプへの書き込みの前に、バッファの状態をチェックして使用可能であることを確認する必要があります。	SIO_put、SIO_get、および SIO_reclaim はブロッキング関数で あり、バッファが使用可能になるまでタスクを待機させます（SIO_issue は非ブロッキング関数です）。
メモリ使用量が少なく、一般に高速です。	柔軟性が高く、一般に使用が簡単です。
各パイプに専用のバッファがあります。	バッファは、コピーをせずに 1 つのストリームから別のストリームへと転送されます（実際にはデータの処理が行われるので、通常は何らかの形でコピー操作が必要です）。
パイプは構成時に静的に作成する必要があります。	ストリームは、実行時に作成することも、構成時に静的に作成することもできます。ストリームは、名前でオープンできます。
デバイスをスタックするためのサポートはビルトインされていません。	デバイスをスタックするためのサポートが提供されています。
HST モジュールをパイプと一緒に使用すると、ホストとターゲット間のデータ転送の取り扱いが簡単になります。	DSP/BIOS には多くのデバイス・ドライバが提供されています。

6.3 ドライバ・モデルの比較

アプリケーション・レベル下で、DSP/BIOS はアプリケーションの DSP ペリフェラルとの通信を可能にする 2 種類のデバイス・ドライバ・モデル（IOM と SIO/DEV）を提供します。

- **IOM モデル。**次の図に、IOM モデルのコンポーネントを示します。これはハードウェアに依存しない層とハードウェアに依存する層に分かれています。クラス・ドライバはハードウェアに依存しません。これはデバイス・インスタンス、入出力（I/O）要求の同期化および順次化を管理します。下位レベルのミニドライバはハードウェアに依存します。IOM モデルは、PIO と DIO アダプタを介してパイプまたはストリームとともに使用できます。IOM モデルの詳細については、『DSP/BIOS Driver Developer's Guide』（文献番号 SPRU616）を参照してください。



- **SIO/DEV モデル。**このモデルはストリーミング入出力（I/O）インターフェイスを提供します。アプリケーションは、SIO モジュールが提供する汎用関数を使用して、ストリームに接続された物理デバイスを管理するデバイス・ドライバにより実装された DEV 関数を間接的に起動します。SIO/DEV モデルはパイプと一緒に使用できません。SIO/DEV モデルの詳細については、第 7 章を参照してください。

両モデルに対して、DEV モジュールを使用してユーザ定義デバイス・オブジェクトを作成します。このデバイスで使用されるモデルは、その関数テーブルのタイプにより識別されます。IOM モデルには IOM_Fxns タイプが使用されます。DEV/SIO モデルには DEV_Fxns タイプが使用されます。

デバイス・オブジェクトは、静的構成を使用するか、または動的に DEV_createDevice 関数を使用して作成することができます。デバイス・オブジェクトを管理するために、DEV_deleteDevice 関数と DEV_match 関数も用意されています。

これ以降では、さまざまな入出力（I/O）ドライバ・オブジェクトとモデルを使用する場合にユーザ定義デバイスを作成する方法について説明します。API 関数呼び出しと構成パラメータの詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』を参照してください。

6.3.1 IOM ミニドライバと使用するデバイスの作成

IOM ミニドライバを GIO クラス・ドライバと使用するには、ユーザ定義デバイスを静的に作成するか、次のような DEV_createDevice 呼び出しを使用して作成します。

```
DEV_Attrs gioAttrs = {
   NULL,              /* device id */
   NULL,              /* device parameters */
   DEV_IOMTYPE,       /* type of the device */
   NULL              /* device global data ptr */

};


status = DEV_createDevice("/codec", &DSK6X_EDMA_IOMFXNS,

                          (Fxn)DSK6X_IOM_init, &gioAttrs);
```

6.3.2 ストリームおよび DIO アダプタと使用するデバイスの作成

IOM ミニドライバを SIO ストリームおよび DIO アダプタと使用するには、ユーザ定義デバイスを静的に作成するか、次のような DEV_createDevice 呼び出しを使用して作成します。

```
DIO_Params dioCodecParams ={
    "/codec",          /* device name */
    NULL               /* chanParams */

};


DEV_Attrs dioCodecAttrs = {
    NULL,              /* device id */
    &dioCodecParams,   /* device parameters */
    DEV_SIOTYPE,       /* type of the device */
    NULL               /* device global data ptr */

};


status =  DEV_createDevice("/dio_codec", &DIO_tskDynamicFxns,

                           (Fxn)DIO_init, &dioCodecAttrs);
```

DEV_createDevice に渡されるドライバ関数テーブルは、タスク（TSK）とともに使用する場合は DIO_tskDynamicFxns、ソフトウェア割り込み（SWI）とともに使用する場合は DIO_cbDynamicFxns です。

6.3.3 SIO/DEV モデルと使用するデバイスの作成

SIO ストリームを SIO/DEV モデルおよび DEV_Fxns 関数テーブル・タイプを使用するデバイス・ドライバと使用するには、ユーザ定義デバイスを静的に作成するか、次のような DEV_createDevice 呼び出しを使用して作成します。

```
DEV_Attrs devAttrs ={
    NULL,               /* device id */
    NULL,              /* device parameters */
    DEV_SIOTYPE,       /* type of the device */
    NULL                /* device global data ptr */

}


status = DEV_createDevice("/codec", &DSK6X_EDMA_DEVFXNS,

                          (Fxn)DSK6X_DEV_init, &devAttrs);
```

DEV_createDevice に渡されるデバイス関数テーブルは、DEV_Fxns タイプです。

6.3.4 付属のソフトウェア・ドライバと使用するデバイスの作成

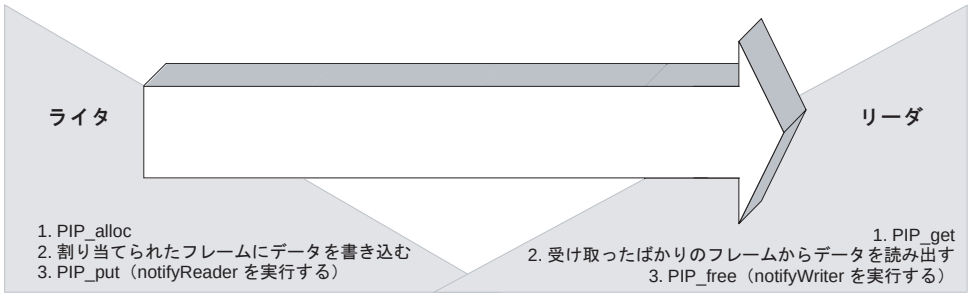
DSP/BIOS には SIO/DEV モデルを使用するいくつかのソフトウェア・ドライバが付属しています。この情報は、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「DEV モジュール」に記載されています。これらのドライバのユーザ定義デバイスの作成は、他の SIO/DEV モデル・ドライバのユーザ定義デバイスの作成と同様です。

6.4 データ・パイプ・マネージャ（PIP モジュール）

パイプは、ブロック入出力（I/O）（ストリーム・ベースの入出力（I/O）または非同期入出力（I/O）ともいう）を管理するように設計されています。各パイプ・オブジェクトは、1 つのバッファを管理します。各バッファは、numframes プロパティと framesize プロパティにより指定される固定数の固定長フレームに分割されています。パイプ上のすべての入出力（I/O）オペレーションでは、一度に 1 フレームが処理されます。各フレームは固定長ですが、アプリケーションは各フレーム内に（そのフレームの長さ以下の範囲で）可変量のデータを入れることができます。

図 6-2 に示すように、パイプには 2 つの端（エンド）があります。ライタ・エンドは、プログラムがデータのフレームを書き込む側です。リーダ・エンドは、プログラムがデータのフレームを読み取る側です。

図 6-2. パイプの両端



データ転送を同期化するために、データ通知関数（notifyReader および notifyWriter）が実行されます。これらの関数は、1 フレームのデータが読み取られたときまたは書き込まれたときにトリガされ、フレームが解放されたことあるいはデータが使用可能であることをプログラムに通知します。これらの関数は、PIP_free または PIP_put を呼び出す関数に関連して実行されます。また、PIP_get または PIP_alloc を呼び出すスレッドからデータ通知関数が呼び出される場合もあります。PIP_get が呼び出されると、DSP/BIOS はパイプ内にフル・フレームが他にもあるかどうかをチェックします。もしあれば、notifyReader 関数が実行されます。PIP_alloc が呼び出されると、DSP/BIOS はパイプ内に空フレームが他にもあるかどうかをチェックします。もしあれば、notifyWriter 関数が実行されます。

1 つのパイプには、単一のリーダと単一のライタが必要です。多くの場合、パイプの一端は HWI で制御され、もう一端はソフトウェア割り込み関数で制御されます。パイプは、プログラム内部の 2 つのアプリケーション・スレッド間でデータを転送するために使用することもできます。

プログラムのスタートアップ処理（詳細は 2.7 節「DSP/BIOS のスタートアップ・シーケンス」（2-13 ページ）を参照）の間に、BIOS_start 関数が DSP/BIOS モジュールをイネーブルにします。スタートアップ時にはすべてのパイプは使用可能な空フレームをもっているので、このステップの一部として、PIP_startup 関数がそれぞれのパイプ・オブジェクトごとに notifyWriter 関数を呼び出します。

パイプで転送するデータには、特別なフォーマット要件やデータ型要件はありません。

DSP/BIOS オンライン・ヘルプでは、データ・パイプ・オブジェクトとそのパラメータについて説明しています。PIP モジュール API については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「PIP Module」を参照してください。

6.4.1 パイプへのデータの書き込み

プログラムでデータをパイプに書き込むときに実行しなければならない手順は、次のとおりです。

1) 最初に、データを入れるために使用可能な空フレームの数をチェックします。そのためには、プログラムで PIP_getWriterNumFrames の返り値をチェックする必要があります。この関数呼び出しは、パイプ・オブジェクト内の空フレームの数を返します。

2) 空フレーム数が 0 より大きい場合は、次に PIP_alloc を呼び出して、パイプから空フレームを 1 つ受け取ります。

3) PIP_alloc 呼び出しから返る前に、DSP/BIOS はパイプ内に他にも使用可能な空フレームがあるかどうかをチェックします。もしあれば、ここで notifyWriter 関数が呼び出されます。

4) PIP_alloc が返ると、アプリケーション・コードは空フレームを使用してデータを格納することができます。これを行うには、関数はフレームの開始アドレスとそのサイズを認識する必要があります。API 関数 PIP_getWriterAddr は、割り当てられたフレームの先頭のアドレスを返します。API 関数 PIP_getWriterSize は、そのフレームに書き込むことができるワード数を返します (空フレームのデフォルト値は、構成に指定されているフレーム・サイズです)。

5) フレームがデータで一杯になると、そのフレームをパイプに返すことができます。フレームに書き込まれるワード数がフレーム・サイズに達しない場合、関数は PIP_setWriterSize 関数を呼び出すとそれを指定することができます。その後で PIP_put を呼び出して、データをパイプに返します。

6) PIP_put を呼び出すと、notifyReader 関数が実行されます。これにより、ライタ・スレッドはパイプ内に読み取り可能なデータがあることをリーダ・スレッドに通知することができます。

図 6-1 の部分コードに、パイプにデータを書き込む方法を示します。

例 6-1.　パイプへのデータの書き込み

```
extern far PIP_Obj writerPipe;   /* created statically */

writer()
{
    Uns size;
    Uns newsize;
    Ptr addr;

    if (PIP_getWriterNumFrames(&writerPipe) > 0) {
        PIP_alloc(&writerPipe);  /* allocate an empty frame */
    }
    else {
        return;      /* There are no available empty frames */
    }

    addr = PIP_getWriterAddr(&writerPipe);
    size = PIP_getWriterSize(&writerPipe);

    ' fill up the frame '

    /* optional */
    newsize = 'number of words written to the frame';
    PIP_setWriterSize(&writerPipe, newsize);

    /* release the full frame back to the pipe */
    PIP_put(&writerPipe);
```

6.4.2　パイプからのデータの読み取り

パイプからフル・フレームを読み取るには、プログラムで次のステップを実行する必要があります。

1) 最初に、パイプから読み取り可能な状態になっているフル・フレームの数をチェックします。そのためには、プログラムで PIP_getReaderNumFrames の返り値をチェックする必要があります。この関数呼び出しは、パイプ・オブジェクト内のフル・フレームの数を返します。

2) フル・フレーム数が 0 より大きい場合は、次に PIP_get を呼び出して、パイプからフル・フレームを 1 つ受け取ります。

3) PIP_get 呼び出しから戻る前に、DSP/BIOS はパイプ内にほかにも使用可能なフル・フレームがあるかどうかをチェックします。もしあれば、ここで notifyReader 関数が呼び出されます。

4) PIP_get が返ると、アプリケーションがフル・フレーム内のデータを読み取ることができるようになります。これを行うには、関数はフレームの開始アドレスとそのサイズを認識する必要があります。API 関数 PIP_getReaderAddr は、フル・フレームの先頭のアドレスを返します。API 関数 PIP_getReaderSize は、そのフレーム内の有効データ・ワード数を返します。

5) アプリケーションがすべてのデータを読み取ると、PIP_free を呼び出すことでフレームをパイプに返すことができます。

6) PIP_free を呼び出すと、notifyWriter 関数が実行されます。これにより、リーダ・スレッドは、パイプ内に使用可能な新しい空フレームがあることをライタ・スレッドに通知することができます。

例 6-2 の部分コードに、パイプからデータを読み取る方法を示します。

例 6-2.　パイプからのデータの読み取り

```
extern far PIP_Obj readerPipe;  /* created statically */

reader()
{
    Uns size;
    Ptr addr;

    if (PIP_getReaderNumFrames(&readerPipe) > 0) {
        PIP_get(&readerPipe);    /* get a full frame */
    }
    else {
        return;       /* There are no available full frames */
    }

    addr = PIP_getReaderAddr(&readerPipe);
    size = PIP_getReaderSize(&readerPipe);

    ' read the data from the frame '

    /* release the empty frame back to the pipe */
    PIP_free(&readerPipe);
}
```

6.4.3　パイプの通知関数の使用

パイプのリーダ・スレッドまたはライタ・スレッドはポーリング・モードで動作できるので、次の順番に当たるフル・フレームまたは空フレームを検索する前に、使用可能なフル・フレームまたは空フレームの数を直接検査することができます。このタイプのポーリング対象の読み取りおよび書き込みオペレーションについては、6.4.1 項「パイプへのデータの書き込み」（6-9 ページ）および 6.4.2 項「パイプからのデータの読み取り」（6-10 ページ）に示す例を参照してください。

ハードウェア・ペリフェラルが書き込む（読み取る）リアルタイム入出力（I/O）ストリームをパイプ・オブジェクトでバッファリングすることによって、そのペリフェラル自体によってトリガされる HWI ルーチンと、最終的にデータを読み取る（書き込む）プログラム関数との間のデータ・チャネルとして、パイプ・オブジェクトを使用する場合がよくあります。このような状況では、パイプの　notifyReader（notifyWriter）関数を構成し、リーダ（ライタ）関数を実行するソフトウェア割り込みを自動的にポストできるようにすることで、アプリケーションと基礎になる入出力（I/O）ストリームとを効率的に同期させることができます。

HWI ルーチンが 1 フレーム分の埋め込み（読み取り）を完了して PIP_put（PIP_free）を呼び出すと、このパイプの通知関数を使用して自動的にソフトウェア割り込みをポストすることができます。この場合、パイプをポーリングしてフレームが使用可能かどうかを調べるのでなく、ソフトウェア割り込みがトリガされたときだけ、つまり読み取り（書き込み）の対象として使用可能なフレームがある場合だけ、リーダ（ライタ）関数が実行されます。

このような関数はデータがレディの場合にのみ呼び出されるので、パイプ内のフレームが使用可能かどうかをチェックする必要はありません。しかし、安全策としてこのような関数でもフレームが使用可能かどうかチェックし、使用可能でなければエラー状態が発生します。次に、そのコード例を示します。

```
if (PIP_getReaderNumFrames(&readerPipe) = 0) {
   error();   /* reader function should not have been posted! */
}
```

このように、パイプ・オブジェクトの通知関数は、他のスレッドおよびハードウェア・デバイスへの入出力（I/O）を管理するためのフロー制御メカニズムとしての機能を果たすことができます。

6.4.4 PIP API の呼び出し順序

各パイプ・オブジェクト内部には、空フレームのリストとパイプのライタ側の空フレーム数を示すカウンタ、およびフル・フレームのリストとパイプのリーダ側のフル・フレーム数を示すカウンタが保持されています。パイプ・オブジェクトには、現在のライタ・フレーム（つまり最後に割り振られてアプリケーションが現在書き込んでいるフレーム）と、現在のリーダ・フレーム（つまりアプリケーションが最後に取得して現在読み取っているフル・フレーム）の記述子も含まれています。

PIP_alloc が呼び出されると、ライタ・カウンタが 1 つ減少します。空フレームが 1 個だけライタ・リストから除去され、このフレームからの情報によってライタ・フレーム記述子が更新されます。アプリケーションがフレームを埋めた後で PIP_put を呼び出すとリーダ・カウンタが 1 つ増加し、DSP/BIOS がライタ・フレーム記述子を使用して新しいフル・フレームをパイプのリーダ・リストに追加します。

注 :

PIP_alloc を再度呼び出すには、事前に PIP_alloc への呼び出しの次に PIP_put への呼び出しを行う必要があります。パイプ入出力（I/O）メカニズムでは、PIP_alloc 呼び出しを連続して行うことはできません。連続コールを行うと、その前の記述子情報が上書きされ、不定の結果が生じることがあります。例 6-3 にその結果を示します。

例 6-3. PIP_alloc の使用

```
/* correct */          /* error! */
PIP_alloc();           PIP_alloc();
...                    ...
PIP_put();             PIP_alloc();
...                    ...
PIP_alloc();           PIP_put();
...                    ...
PIP_put();             PIP_put();
```

同様に、PIP_get が呼び出されると、リーダ・カウンタが 1 つ減少します。フル・フレームが 1 個だけリーダ・リストから除去され、このフレームからの情報によってリーダ・フレーム記述子が更新されます。アプリケーションがフレームを読み取った後で PIP_free を呼び出すとライタ・カウンタが 1 つ増加し、DSP/BIOS がリーダ・フレーム記述子を使用して新しい空フレームをパイプのライタ・リストに追加します。PIP_get を再度呼び出すには、例 6-4 に示すように、事前に PIP_get への呼び出しの次に PIP_free への呼び出しを行う必要があります。

パイプ入出力（I/O）メカニズムでは、PIP_get 呼び出しを連続して行うことはできません。連続呼び出しを行うと、その前の記述子情報が上書きされ、不定の結果が生じることがあります。

例 6-4. PIP_get の使用

```
/* correct */          /* error! */
PIP_get();             PIP_get();
...                    ...
PIP_free();            PIP_get();
...                    ...
PIP_get();             PIP_free();
...                    ...
PIP_free();            PIP_free();
```

6.4.4.1 再帰に関する問題の回避

パイプの通知関数が同じパイプ用の PIP API を呼び出す場合は、注意が必要です。

たとえば、パイプの notifyReader 関数が同じパイプ用の PIP_get を呼び出します。パイプのリーダは HWI ルーチンです。パイプのライタは SWI ルーチンです。したがって、リーダの方がライタより優先順位が高くなります（この例では、notifyReader から PIP_get を呼び出すことは道理にかなっています。こうすることで次のフル・バッファが使用可能になり次第、アプリケーションがそのフル・バッファを準備してハードウェア割り込みが再度トリガされる前に、リーダ（HWI ルーチン）がそれを使用可能な状態にできるからです）。

パイプのリーダ関数 HWI ルーチンはパイプからデータを読み取るために PIP_get を呼び出します。パイプのライタ関数 SWI ルーチンは PIP_put を呼び出します。notifyReader への呼び出しは現在のルーチンに関連して PIP_put 内で発生するため、PIP_get への呼び出しも SWI ライタ・ルーチンから発生します。

したがって、ここに示す例では、優先順位の異なる 2 つのスレッドが同じパイプ用の PIP_get を呼び出すことになります。片方のスレッドが他方のスレッドより優先された結果、PIP_free をコ呼び出す前に PIP_get が 2 回呼び出された場合、または異なるスレッドからの同じパイプに関する呼び出しのときに PIP_get が優先されて再度呼び出された場合には、破滅的な結果を招くことがあります。

注：

再帰を回避するための一般的な規則は、PIP 関数を notifyReader および notifyWriter の一部として呼び出さないようにすることです。アプリケーションの効率を向上させるためにこの呼び出しが必要な場合は、これを保護対象にして同一パイプ・オブジェクトへの再入可能性を回避し、PIP API の誤った呼び出し順序の発生を防止する必要があります。

6.5 メッセージ・キュー

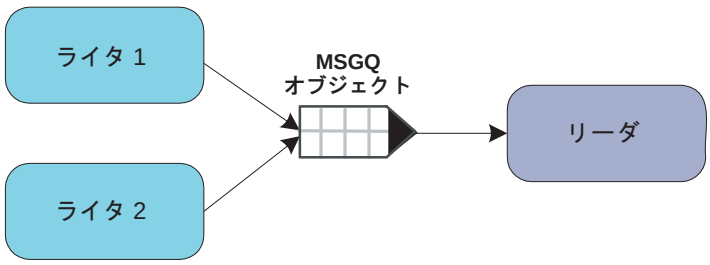
MSGQ モジュールは、可変長メッセージの構造化された送受信をサポートします。このモジュールは、同種または異種のマルチプロセッサ・メッセージングに使用できます。特に OMAP デバイスに含まれる特定の TI 汎用プロセッサ（GPP）の DSP/BIOS Link に実質的に同等の MSGQ API が実装されています。

MSGQ は他のモジュールよりも高度なメッセージングを提供します。これは通常マルチプロセッサ・メッセージングなどの複合的な状況で使用されます。MSGQ モジュールの重要な機能を次に示します。

- ライタとリーダはルーチン・コードを変更することなく別のプロセッサに再配置されます。
- メッセージの受信時のタイムアウトが可能です。
- リーダはライタと応答を決定できます。
- タイムアウトがゼロの場合、メッセージの受信は決定論です。
- メッセージの送信は決定論です（呼び出し、ただしデリバリではありません）。
- メッセージは任意のメッセージ・キューに常駐できます。
- ゼロコピー転送をサポートします。
- HWI、SWI、および TSK から送受信できます。
- アプリケーションで通知メカニズムが指定されます。
- メッセージ・バッファ・プールでの QoS（Quality of Service）が可能です。たとえば、特定のメッセージ・キューに対して特定のバッファ・プールを使用します。

メッセージはメッセージ・キューを介して送受信されます。リーダはメッセージ・キューからメッセージを取得する（読み取る）スレッドです。ライタはメッセージ・キューにメッセージを入れる（書き込む）スレッドです。各メッセージ・キューに 1 つのリーダがあり、多数のライタを保持することが可能です。スレッドは複数のメッセージ・キューとの間で読み取りまたは書き込みを行います。

図 6-3. メッセージ・キューのライタおよびリーダ



概念的に、リーダ・スレッドはメッセージ・キューを所有します。このリーダ・スレッドはメッセージ・キューをオープンします。ライタ・スレッドは既存のメッセージ・キューを探して、それらを利用します。

メッセージは MSGQ モジュールから割り当てられなければなりません。メッセージが割り当てられると、任意のメッセージ・キューに送信できます。メッセージが送信されると、ライタはメッセージの所有権を失い、そのメッセージを変更できません。リーダはメッセージを受信すると、そのメッセージを所有します。リーダはそのメッセージを解放するか、あるいは再利用します。

メッセージ・キュー内のメッセージは可変長にすることができます。唯一の要件として、メッセージの定義にある最初のフィールドは MSGQ_MsgHeader 要素でなければなりません。

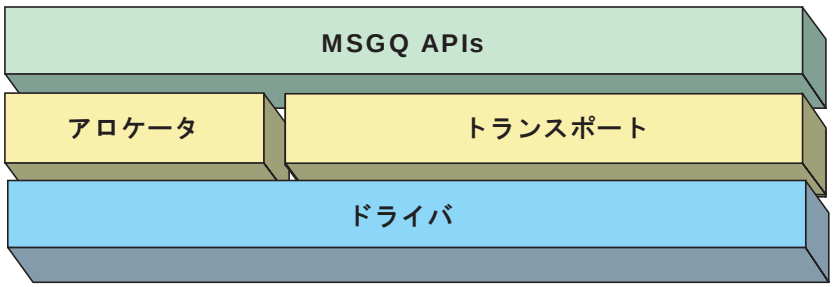
```
typedef struct MyMsg {
    MSGQ_MsgHeader header;
    ...
} MyMsg;
```

MSGQ API は MSGQ_MsgHeader を内部で使用します。ご使用のアプリケーションでは、MSGQ_MsgHeader のフィールドを変更したり、直接アクセスしたりしてはいけません。

MSGQ モジュールのコンポーネントは、次のとおりです。

- **MSGQ API。**アプリケーションは、メッセージを送受信するために MSGQ 関数を呼び出してメッセージ・キュー・オブジェクトをオープンして使用します。概要については、「MSGQ API」（6-17 ページ）を参照してください。詳細については、各 API の説明を参照してください。
- **アロケータ。**MSGQ を介して送信されたメッセージは、アロケータによって割り当てられなければなりません。アロケータはメッセージのメモリが割り当てられる場所および方法を決定します。アロケータの詳細については、「アロケータ」（6-20 ページ）を参照してください。
- **トランスポート。**トランスポートは他のプロセッサとともにメッセージの特定と送信を行います。トランスポートの詳細については、「トランスポート」（6-21 ページ）を参照してください。

図 6-4.　MSGQ アーキテクチャのコンポーネント



アロケータとトランスポートには標準のインターフェイスがあります。アロケータとトランスポートのインターフェイス関数は、アプリケーションではなく MSGQ 関数により呼び出されます。DSP/BIOS では簡単な静的アロケータを用意していますが、他のアロケータとトランスポートを標準のインターフェイスに従って実装することができます。

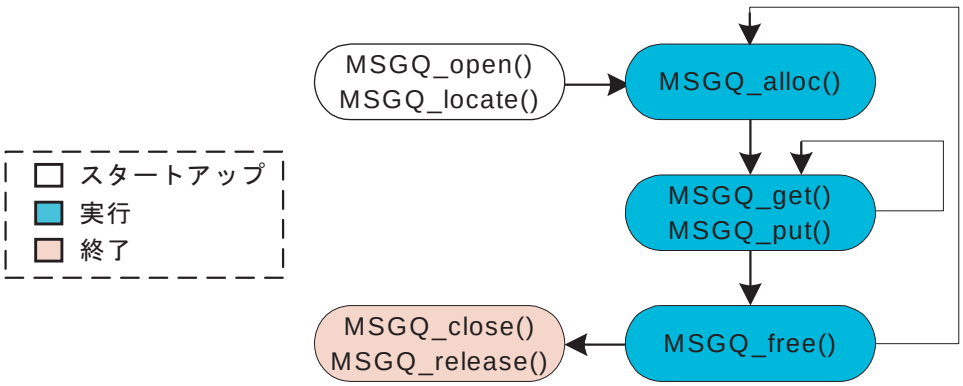
注：本書ではアロケータまたはトランスポートの作成方法については説明していません。アロケータとトランスポートの設計については、将来的に提供する予定です。

6.5.1 MSGQ API

MSGQ API はメッセージ・キューのオープンとクローズ、およびメッセージの送受信を行うために使用します。MSGQ API により、アプリケーションはトランスポートとアロケータに関する知識をもつ必要がありません。

次の図に、主な MSGQ 関数の呼び出しシーケンスを示します。

図 6-5. MSGQ 関数の呼び出しシーケンス



リーダは次の API を呼び出します。

- MSGQ_open
- MSGQ_get
- MSGQ_free
- MSGQ_close

ライタは次の API を呼び出します。

- MSGQ_locate または MSGQ_locateAsync
- MSGQ_alloc
- MSGQ_put
- MSGQ_release

決定的実行時間を保持するために、可能な限り MSGQ API が書き込まれていました。これにより、アプリケーション設計者はメッセージングが不明な数のサイクルを消費しないことを確認できます。

また、MSGQ 関数はすべてのタイプの DSP/BIOS スレッド（HWI、SWI、および TSK）からのメッセージ・キューの使用をサポートします。すなわち、同期（ブロッキング）となる呼び出しは非同期（非ブロッキング）の代替を保持します。

6.5.2 静的構成

MSGQ モジュールとそれが依存するアロケータを使用するには、次の項目を静的に構成する必要があります。

- ❏ アプリケーション・コードの MSGQ_config 変数（下記を参照）
- ❏ Tconf を使用する MSGQ モジュールの ENABLEMSGQ プロパティ
- ❏ Tconf を使用する GBL モジュールの PROCID プロパティ
- ❏ Tconf を使用する POOL モジュールの ENABLEPOOL プロパティ
- ❏ アプリケーション・コードの POOL_config 変数

上記のプロパティの設定については、ご使用のプラットフォームに対応した『DSP/BIOS Application Programming Interface Guide』を参照してください。

MSGQ モジュールを使用するために、アプリケーションは値のセットされた MSGQ_config 変数を指定しなければなりません。

```
MSGQ_Config MSGQ_config;
```

MSGQ_Config タイプの構造は次のとおりです。

```
typedef struct MSGQ_Config {
   MSGQ_Obj          *msgqQueues;     /* Array of message queue handles */
   MSGQ_TransportObj *transports;     /* Array of transports */
   Uint16            numMsgqQueues;   /* Number of message queue handles*/
   Uint16            numProcessors;   /* Number of processors */
   Uint16            startUninitialized;  /* First msgq to init */
   MSGQ_Queue        errorQueue;      /* Receives async transport errors*/
   Uint16            errorPoolId;     /* Alloc error msgs from poolId */
} MSGQ_Config;
```

次の表に、MSGQ_Config 構造内のフィールドを示します。

フィールド	タイプ	説明
msgqQueues	MSGQ_Obj *	メッセージ・キュー・オブジェクトの配列。各オブジェクトのフィールドは初期化する必要がありません。

フィールド	タイプ	説明
transports	MSGQ_TransportObj *	トランスポート・オブジェクトの配列。各オブジェクトのフィールドは初期化する必要があります。
numMsgqQueues	Uint16	msgqQueues 配列の長さ。
numProcessors	Uint16	transports 配列の長さ。
startUninitialized	Uint16	msgqQueue 配列で初期化するための最初のメッセージ・キューのインデックス。これは 0 にセットする必要があります。
errorQueue	MSGQ_Queue	トランスポート・エラーを受信するためのメッセージ・キュー。MSGQ_INVALIDMSGQ に初期化します。
errorPoolId	Uint16	トランスポート・エラーを割り当てるためのアロケータ。POOL_INVALIDID に初期化します。

MSGQ は内部で MSGQ_config 変数を介してその構成を参照します。MSGQ モジュールが（Tconf を使用して）イネーブルであるのに、アプリケーションが MSGQ_config 変数を提供しない場合、アプリケーションは正常にリンクされません。

MSGQ_Config 構造体では、MSGQ_TransportObj 項目の配列によりトランスポート・オブジェクトが次の構造体で定義されます。

```
typedef struct MSGQ_TransportObj {
  MSGQ_MqtInit  initFxn;    /* Transport init func */
  MSGQ_TransportFxns *fxns; /* Interface funcs */
  Ptr         params; /* Setup parameters */
  Ptr         object; /* Transport-specific object */
  Uint16      procId; /* Processor Id talked to */
} MSGQ_TransportObj;
```

次の表に、MSGQ_TransportObj 構造体内のフィールドを示します。

フィールド	タイプ	説明
initFxn	MSGQ_MqtInit	このトランスポートのための初期化関数。この関数は DSP/BIOS のスタートアップ時に呼び出されます。より明確に述べると、これは main() の前に呼び出されます。
fxns	MSGQ_TransportFxns *	トランスポートのインターフェイス関数へのポインタ。
params	Ptr	トランスポートのパラメータへのポインタ。このフィールドはトランスポート固有です。このフィールドについては、トランスポートに付属の資料を参照してください。

フィールド	タイプ	説明
info	Ptr	トランスポートに必要な状態情報。このフィールドはトランスポートで初期化されて管理されます。このフィールドの使用方法を判別するには、特定のトランスポート実装を参考にしてください。
procId	Uint16	このトランスポートが通信するプロセッサの数値 ID。現行のプロセッサには、GBL.PROCID プロパティと一致する procId フィールドが必要です。

MSGQ_TransportObj でパラメータ構造体が指定されていない場合（すなわち、MSGQ_NOTRANSPORT が使用されている場合）、トランスポートはデフォルト・パラメータを使用します。

シングルプロセッサ・アプリケーションの MSGQ 構成例を次に示します。

```
#define NUMMSGQUEUES  4  /* # of local message queues*/
#define NUMPROCESSORS 1 /* Single processor system */

static MSGQ_Obj           msgQueues[NUMMSGQUEUES];
static MSGQ_TransportObj  transports[NUMPROCESSOR] =
                            {MSGQ_NOTRANSPORT};

MSGQ_Config MSGQ_config = {
   msgQueues,
   transports,
   NUMMSGQUEUES,
   NUMPROCESSORS,
   0,
   MSGQ_INVALIDMSGQ,
   POOL_INVALIDID
};
```

6.5.3 アロケータ

MSGQ モジュールを介して送信されるメッセージは、アロケータによって割り当てられなければなりません。アロケータはメッセージのメモリが割り当てられる場所および方法を決定します。

アロケータはアロケータ・インターフェイスの実装のインスタンスです。アプリケーションはアロケータのインスタンスを 1 つ以上作成します。

POOL モジュールは、アロケータが指定しなければならない標準インターフェイス関数を記述します。アロケータ・インターフェイス関数は、ユーザ・アプリケーションではなく MSGQ モジュールにより内部で呼び出されます。DSP/BIOS では STATICPOOL と呼ばれるわかりやすい静的アロケータを用意していますが、他のアロケータを標準のインターフェイスに従って実装することができます。

注：本書ではアロケータの作成方法については説明していません。アロケータとトランスポートの設計については、将来的に提供する予定です。

アプリケーションでは複数のアロケータを使用できます。複数のアロケータを使用することで、アプリケーションはメッセージの使用を規制することができます。たとえば、高速オンチップ・メモリのプールから重要なメッセージを割り当て、低速な外部メモリの別のプールから重要でないメッセージを割り当てることができます。

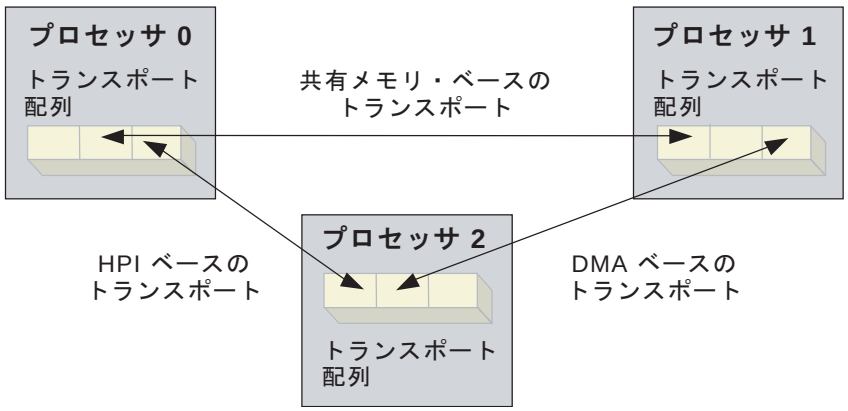
6.5.4 トランスポート

トランスポートの役割は、他のプロセッサへの物理リンクを超えて通信することです。トランスポート・インターフェイスにより、ユーザはアプリケーションを変更せずに（トランスポートの構成を除く）、基本的な通信メカニズムを変更できます。

トランスポートはトランスポート・インターフェイスの実装のインスタンスです。各プロセッサにトランスポートの配列があります。2 つのプロセッサ間に最大 1 つのトランスポートが存在します。この配列はプロセッサ ID に基づきます。したがって、各プロセッサのトランスポート配列の最初の要素（0 番目のインデックス）にプロセッサ 0 へのトランスポートが含まれます。プロセッサ 0 では、最初の要素は未使用の MSGQ_NOTRANSPORT トランスポートです。

たとえば、次の図に示されているシステムを検討してください。このシステムでは、3 つのプロセッサが DSP/BIOS を実行しています。トランスポートは矢印で示されています。

図 6-6. マルチプロセッサのトランスポート例



システム内には 3 つのプロセッサが存在するため、各プロセッサに 3 つのトランスポートの配列があります。

プロセッサ	トランスポート配列
プロセッサ 0	[0]: MSGQ_NOTRANSPORT トランスポート [1]: 共有メモリに基づいたプロセッサ 1 へのトランスポート [2]: HPI に基づいたプロセッサ 2 へのトランスポート
プロセッサ 1	[0]: 共有メモリに基づいたプロセッサ 0 へのトランスポート [1]: MSGQ_NOTRANSPORT トランスポート [2]: DMA に基づいたプロセッサ 2 へのトランスポート
プロセッサ 2	[0]: HPI に基づいたプロセッサ 0 へのトランスポート [1]: DMA に基づいたプロセッサ 1 へのトランスポート [2]: MSGQ_NOTRANSPORT トランスポート

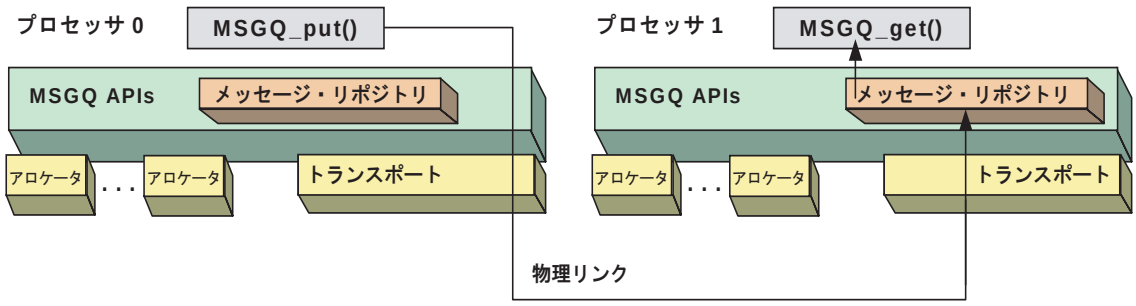
この例では、現行のプロセッサに対応する配列位置に MSGQ_NOTRANSPORT を使用しています。また、システム内で 2 つの別々のプロセッサが通信していない場合、トランスポート配列の適切な位置で MSGQ_NOTRANSPORT を使用する必要があります。

MSGQ モジュールは、トランスポートが指定しなければならない標準インターフェイス関数を記述します。トランスポート・インターフェイス関数は、ユーザ・アプリケーションではなく MSGQ 関数により呼び出されます。

注：本書ではトランスポートの作成方法については説明していません。アロケータとトランスポートの設計については、将来的に提供する予定です。

トランスポート間のプロトコルはトランスポート固有ですが、メッセージ・キューを特定して、物理境界を越えてメッセージを送信できる必要があります。次の図に、メッセージを別のプロセッサのメッセージ・キューに送信する例を示します。

図 6-7.　リモート・トランスポート



6.5.5 マルチプロセッサについて

MSGQ モジュールの主な機能の 1 つは、マルチプロセッサ環境での透過性です。リーダをあるプロセッサから別のプロセッサに移動しても、ライタ・コードに変更はありません。同様に、ライタを別のプロセッサに移動しても、リーダに変更はありません。

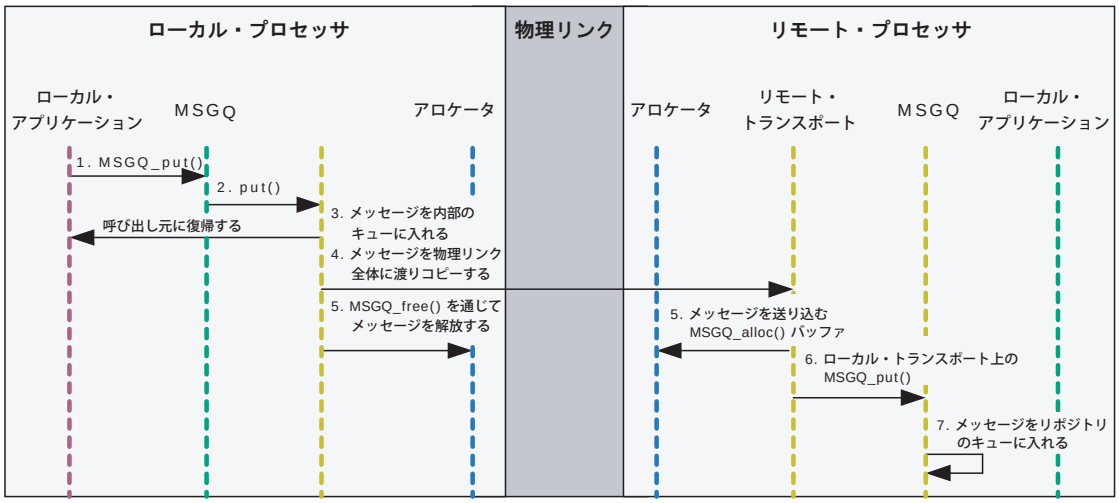
各プロセッサでは、他のプロセッサごとに 1 つのトランスポートが存在します。たとえば、相互に通信する 4 つのプロセッサを搭載するシステムでは、各プロセッサに 3 つのトランスポート（1 つは他の 3 つの各プロセッサと対話）と 1 つの MSGQ_NOTRANSPORT トランスポートが必要です。2 つのプロセッサ間に複数の物理リンクがある場合（共有メモリとシリアルなど）、トランスポートが 2 つのリンクを管理する必要があります（どのメッセージをどのリンクに送るかの判別など）。

メッセージが割り当てられると、アロケータの ID が割り当てられたメッセージの MSGQ_MsgHeader 部分に埋め込まれます。これにより、メッセージを後で解放しやすくなります。すなわち、アプリケーションはメッセージを割り当てる際に使用したアロケータを記憶しておく必要がありません。

一部のトランスポートはコピー・ベースです。このようなトランスポートがメッセージをリモート・プロセッサに送信する場合、そのメッセージは物理リンクにコピーされます（TCP/IP など）。コピー・ベースのトランスポートの場合、ソース・プロセッサのトランスポートがメッセージをそのコピー後に解放します。宛先プロセッサのトランスポートは、メッセージを割り当てて、それを宛先のメッセージ・キューに送信します。トランスポートの割り当てと解放はすべてアプリケーションに対して透過的です。しかし、リーダ・スレッドは受信されたメッセージの解放または再利用を行う必要があります。

次の図に、メッセージをコピー・ベースのトランスポートを介してリモート・プロセッサに送信する際に発生するイベントのシーケンスを示します。メッセージの受信は、3 つのイベント後にリモート・プロセッサで MSGQ_get を介して行います。

図 6-8.　リモート・プロセッサへのメッセージの送信時に発生するイベント



ゼロコピー・ベースのトランスポートの場合（共有メモリの使用など）、トランスポートは単にメッセージが存在することを他方に信号で通知します。この場合、中間にある割り当てや解放はありません。しかし、リーダは受信したメッセージの解放または再利用を行う必要があります。

いずれのタイプのトランスポートでも、異なるプロセッサにおけるアロケータ構成は同一でなければなりません。次の 2 つの例を参照してください。

- **ゼロコピー・ベースのリモート・トランスポート。**アロケータ 0 がプロセッサ A で共有メモリのアロケータである場合、プロセッサ B のアロケータ 0 は同一の共有メモリで動作する必要があります。

- **コピー・ベースのリモート・トランスポート。**アロケータ 1 が 64 バイトのメッセージを割り当てる場合、プロセッサ B のアロケータ 1 も 64 バイトのメッセージを割り当てる必要があります（メッセージが両方向に流れている場合）。基本的な割り当てメカニズムは異なっても、メッセージのサイズは同一でなければなりません。

ルーティングにより、メッセージをあるプロセッサから別のプロセッサに中間プロセッサを介して送信することができます。2 つのプロセッサ間に物理リンクがない場合に、ルーティングが必要になります。ルーティングは、MSGQ モジュールで直接サポートされていません。ルーティングは MSGQ モジュールの一番上に構成されますが、アプリケーション・レベルで管理する必要があります。たとえば、ユーザがルータとして動作するスレッドを作成します。

MSGQ またはトランスポートのいずれもメッセージのユーザ部分でエンディアン変換を実行しません。トランスポートはメッセージの MSGQ_MsgHeader 部分では必要なエンディアン変換を実行しますが、メッセージの他の部分では実行しません。アプリケーションがメッセージの他の部分でのエンディアン変換を管理する必要があります。

6.5.6 データ転送モジュールの比較

DSP/BIOS でのデータ移動には、一部のモジュールが使用可能です。

- MBX。メールボックス・モジュール。
- MSGQ。メッセージ・キュー・モジュール。
- PIP。パイプ・モジュール。
- QUE。キュー・モジュール。
- SIO。ストリーミング入出力（I/O）モジュール。

SIO と PIP は両方ともストリーミング・モデルを使用します。ここでは、DSP/BIOS から見たストリーミングとメッセージングの違いを示します。

- ストリームは、リアルタイム・データの連続したシーケンスです。ストリーミングは 1 つのライタと 1 つのリーダのポイント・ツー・ポイントです。これは通常ゼロコピーとともに実行されます。
- メッセージは、非同期の制御情報です。メッセージングは通常複数のライタと 1 つのリーダとともに実行されます。

MSGQ、QUE、および MBX には、いくつか異なる点があります。それぞれのモジュールには長所と短所があり、最終的にアプリケーション設計者がアプリケーションに最適なモジュールを決定する必要があります。最適なモジュールを決定する際に考慮する点を次に示します。

- **マルチプロセッサのサポート。**MSGQ はマルチプロセッサをサポートします。QUE または MBX はいずれもマルチプロセッサをサポートしません。
- **メッセージの所有権。**メッセージが MSGQ_put または QUE_put で送信されると、メッセージの所有権が放棄されます。メッセージを受信すると、リーダは所有権を取得します。MBX モジュールでは、メッセージを MBX_post で送信すると、呼び出しが返る前にメッセージが内部で MBX にコピーされます。したがって、MBX_post が返ると、送信元が引き続きバッファを制御します。
- **メッセージのコピー。**MBX モジュールはコピー・ベースです。QUE はゼロコピーです。MSGQ の場合、プロセッサ内の転送はゼロコピー動作です。プロセッサ間の転送はコピー・ベースの場合も、そうでない場合もあります（トランスポートに依存）。
- **通知メカニズム。**MSGQ と MBX は通知メカニズムを提供します。このため、リーダはメッセージの待機中にブロックすることができます。さらに、MSGQ ではユーザが通知メカニズムを指定できます（MBX では常にセマフォです）。したがって、通知は SWI をポストする可能性があります。QUE には通知メカニズム

のタイプはありません。アプリケーションがポーリングしたりセマフォを使用したりして、これを処理する必要があります。

- **メッセージのサイズと数。**MBX モジュールには、メールボックスごとに固定長のメッセージとメッセージの数があります。これらの値はメールボックス作成時に指定されます。QUE と MSGQ では、可変長のメッセージが可能です。これらには受信を待機するメッセージの最大数はありません。
- **複雑さと占有スペース。**MSGQ モジュールは多くの高度な機能を備えています。しかし、これにより複雑さが増し、占有スペースが多くなります。このような高度な機能と柔軟性を必要としないアプリケーションでは、QUE または MBX モジュールを使用する方がその少ない占有スペースと使いやすさという点でより適した方法です。

6.6 ホスト・チャネル・マネージャ（HST モジュール）

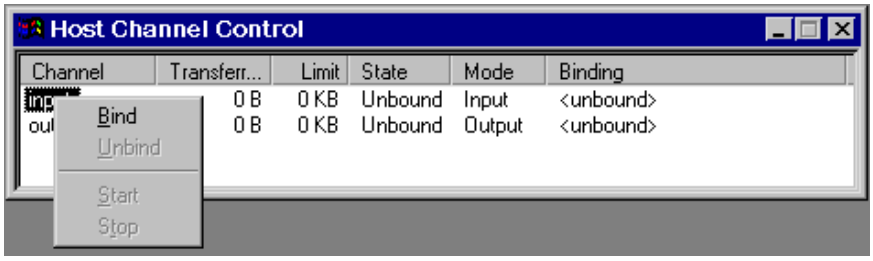
HST モジュールはホスト・チャネル・オブジェクトを管理し、これによりアプリケーションはターゲットとホスト間でストリーム方式でデータを転送することができます。ホスト・チャネルは、入力用または出力用に構成されています。入力ストリームは、ホストからターゲットへのデータを読み取ります。出力ストリームは、ターゲットからホストへデータを転送します。

注：

HST チャネル名は、先頭文字を下線（ _ ）で書き始めることはできません。

Code Composer Studio の「Host Channel Control」を右クリックすると、チャネルを PC ホスト上のファイルに動的にバインドすることができます。その後に、例 6-9 に示すように、各チャネルでデータ転送を開始することができます。

図 6-9. チャネルのバインド



各ホスト・チャネルは、内部的にはパイプ・オブジェクトを使用して実装されています。特定のホスト・チャネルを使用するために、プログラムは HST_getpipe を使用して該当のパイプ・オブジェクトを取得します。次に、入力の場合は PIP_get と PIP_free オペレーションを呼び出すことで、出力の場合は PIP_alloc と PIP_put オペレーションを呼び出すことでデータを転送します。

データを読み取るためのコードは、例 6-5 のようになります。

例 6-5.　ホスト・チャネルを介したデータの読み取り

```
extern far HST_Obj input;

readFromHost()
{
    PIP_Obj *pipe;
    Uns size;
    Ptr addr;

   pipe = HST_getpipe(&input)   /* get a pointer to the host
                                   channel's pipe object */
    PIP_get(pipe);              /* get a full frame from the
                                      host */
    size = PIP_getReaderSize(pipe);
    addr = PIP_getReaderAddr(pipe);

    ' read data from frame '

   PIP_free(pipe);     /* release empty frame to the host */
}
```

各ホスト・チャネルは、入力チャネル用の 1 フレーム分のデータ（または出力チャネル用のフリー・スペース）が使用可能なときに実行するデータ通知関数を指定することができます。ホストが 1 フレームのデータの書き込みまたは読み取りを行ったときに、この関数がトリガされます。

HST チャネルは、プラットフォームに応じて、ファイルを 16 ビットまたは 32 ビット・ワードから構成される生データとして取り扱います。データのフォーマットはアプリケーション固有なので、ホストとターゲットのデータ・フォーマットと配列が一致していることを確認する必要があります。たとえば、ホストから 32 ビットの整数を読み取っている場合は、正しいバイト順序でデータが含まれているファイルをホスト・ファイルとして使用する必要があります。バイト順序が正しいことの他には、ホストとターゲット間で転送するデータに必要なフォーマットやデータ型の要件は特にありません。

プログラム開発時に、HST オブジェクトを使用してデータ・フローをシミュレートし、プログラム・アルゴリズムによってデータに加えられた変更をテストすることができます。開発の初期段階、特に信号処理アルゴリズムをテストするときには、プログラムで入力チャネルを明示的に使用して、そのアルゴリズムへの入力元のファイルのデータ・セットにアクセスし、出力チャネルを使用してアルゴリズム出力を記録します。出力ホスト・チャネルを経てファイルに保存されたデータを予期していた結果と比較することで、アルゴリズムのエラーを検出することができます。プログラム開発サイクルの後期にアルゴリズムが正常と判断された時点で、HST オブジェクトを、実動ハードウェア用の他のスレッドまたは入出力（I/O）ドライバと通信する PIP オブジェクトに変更することができます。

6.6.1 ホストへの HST データの転送

データ・ストリームをホストにリアルタイムで転送するために使用可能な帯域幅の量は、最終的には物理データ・リンクの選択により決まるのに対して、HST Channel インターフェイスは物理リンクから常に独立しています。HST マネージャを使用すると、使用可能な複数の物理接続の中から接続を選択することができます。

ホストへの実際のデータ転送は、C54x プラットフォームの場合は、アイドル・ループに入っている間に最も低い優先順位で実行されます。

C55x および C6000 プラットフォームの場合は、ホスト PC がターゲットとの間でデータを転送するための割り込みをトリガします。この割り込みには、SWI、TSK、および IDL 関数より高い優先順位が割り当てられています。実際の ISR 関数は非常に短時間のうちに実行されます。アイドル・ループの中では、LNK_dataPump 関数は RTDX バッファを準備し、RTDX 呼び出しを実行するという、さらに時間のかかる作業を行います。高い優先順位で実行されるのは、実際のデータ転送だけです。特に大量の LOG データを転送する必要がある場合は、このデータ転送によりわずかながらリアルタイムの動作が影響を受けることがあります。

6.7 入出力（I/O）のパフォーマンスについて

HST オブジェクトを使用している場合、ホスト PC は、LNK_dataPump オブジェクトにより指定された関数を使用してデータを読み取ったり書き込んだりします。これはビルトイン IDL オブジェクトで、その関数をバックグラウンド・スレッドの一部として実行します。バックグラウンド・スレッドの優先順位は最も低いので、C54x プラットフォームでは、ソフトウェア割り込みおよびハードウェア割り込みの方がデータ転送より優先されます。これに対して C55x および C6000 プラットフォームでは、実際のデータ転送が高い優先順位で行われます。

LOG、STS、および TRC でセットしたポーリング・レートは、HST オブジェクトのデータ転送速度を制御しません。LOG、STS、および TRC データも転送する必要があるので、実際にはポーリング・レートが高速になるとデータ転送速度がやや遅くなります。

第 7 章

ストリーム入出力（I/O）とデバイス・ドライバ

本章では、DEV_Fxns モデルを使用するデバイス・ドライバの作成と使用に関係する事項について説明し、合わせてプログラミング例をいくつか示します。

項目 **ページ**

7.1 ストリーム入出力（I/O）と デバイス・ドライバの概要

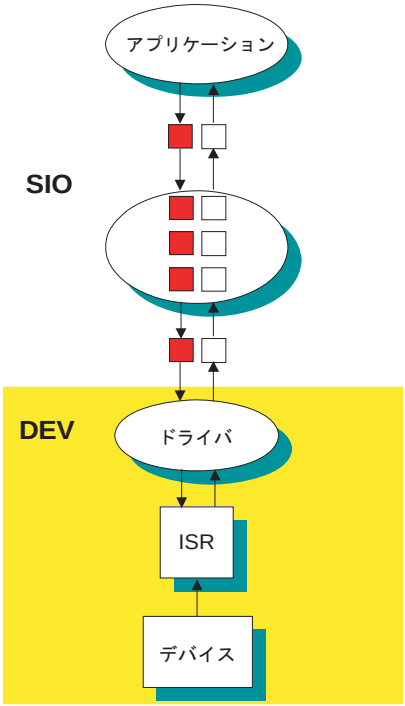
注：

本章では、DEV_Fxns 関数テーブル型を使用するデバイスについて説明します。『DSP/BIOS Driver Developer's Guide』（文献番号 SPRU616）では、新しいデバイス・ドライバ・モデル、すなわち IOM_Fxns 型の関数テーブルを使用する IOM モデルについて解説します。IOM ミニドライバの作成と IOM ミニドライバをアプリケーションに統合する方法については、該当の資料を参照してください。

SIO ストリームの使用に関する本章の情報は、SIO ストリームを IOM ミニドライバとともに使用する場合に関するものです。

第 6 章では、DSP/BIOS でサポートされているデバイスから独立した入出力（I/O）オペレーションについて、アプリケーション・プログラムの視点から説明しました。プログラムは、SIO モジュールが提供する汎用関数を使用して、ストリームに接続された特定の物理デバイスを管理するドライバにより実装された該当する関数を間接的に起動します。図 7-1 の網掛け部分に示すように、本章では、このドライバからこのインターフェイスを見た場合の DSP/BIOS におけるデバイスに依存しない入出力 （I/O）について説明します。

図 7-1. DSP/BIOS のデバイスに依存しない入出力（I/O）



他のモジュールと異なり、アプリケーション・プログラムは、SIO モジュールが管理する個々のデバイス・オブジェクトを操作するドライバ関数を直接的には呼び出しません。代わりに各ドライバ・モジュールは、特別の名前が付けられた特定の型の構造体（DEV_Fxns）をエクスポートします。SIO モジュールはこの構造体を使用して、汎用関数呼び出しを適切なドライバ関数に送ります。

表 7-1 に示すように、SIO オペレーションでは、このテーブルを参照して該当のドライバ関数が呼び出されます。Dxx は、ユーザが特定のデバイス用に作成するデバイス固有関数を示します。

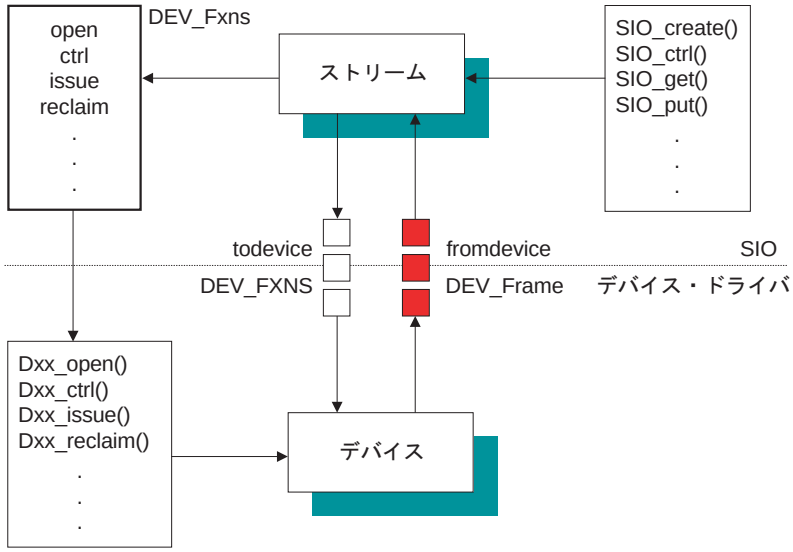
表 7-1.　内部ドライバ・オペレーションに対する汎用入出力（I/O）オペレーション

汎用入出力（I/O）オペレーション	内部ドライバ・オペレーション
SIO_create(name, mode, bufsize, attrs)	Dxx_open(device, name)
SIO_delete(stream)	Dxx_close(device)
SIO_get(stream, &buf)	Dxx_issue(device) および Dxx_reclaim(device)
SIO_put(stream, &buf, nbytes)	Dxx_issue(device) および Dxx_reclaim(device)
SIO_ctrl(stream, cmd, arg)	Dxx_ctrl(device, cmd, arg)
SIO_idle(stream)	Dxx_idle(device, FALSE)
SIO_flush(stream)	Dxx_idle(device, TRUE)
SIO_select(streamtab, n, timeout)	Dxx_ready(device, sem)
SIO_issue(stream, buf, nbytes, arg)	Dxx_issue(device)
SIO_reclaim(stream, &buf, &arg)	Dxx_reclaim(device)
SIO_staticbuf(stream, &buf)	なし

これらの内部ドライバ関数は、カーネルのマルチタスク機能からアプリケーション・レベルのサービスに至るまで、DSP/BIOS が提供する事実上すべての機能に依存しています。ドライバは、DSP/BIOS のデバイスに依存しない入出力（I/O）インターフェイスを使用して、主としてスタック可能デバイスをサポートすることを目的として他のドライバと間接的に通信します。

図 7-2 に、デバイス、Dxx デバイス・ドライバ、およびデバイスからデータを受け入れるストリーム の相互関係を示します。SIO は、DEV_Fxns（該当デバイス用の関数テーブル）にリストされている Dxx 関数を呼び出します。入力ストリームと出力ストリームは、どちらも極小キュー device→todevice および device→fromdevice を使用してデバイスとの間でバッファを交換します。

図 7-2.　デバイス、ドライバ、およびストリームの関係



各デバイス・ドライバについて、それぞれ Dxx_open、Dxx_idle、Dxx_input、Dxx_output、Dxx_close、Dxx_ctrl、Dxx_ready、Dxx_issue、および Dxx_reclaim を記述する必要があります。

7.2 ストリームの作成と削除

アプリケーションで特定のデバイスとの間のストリーム入出力（I/O）処理を実行できるようにするには、まずそのデバイスを構成に追加しておく必要があります。製品配布パッケージに含まれている任意のドライバ用、またはユーザ提供のドライバ用のデバイスを追加することができます。ストリームを使用して特定のデバイスとの間で入出力（I/O）を行うには、まずそのデバイスを構成します。次に、構成時にまたは実行時に SIO_create 関数を使用して、ストリーム・オブジェクトを作成します。

7.2.1 ストリームの静的な作成

構成時に、ストリームを作成し、個々のストリームおよび SIO マネージャ自体についてプロパティをセットすることができます。静的に作成したストリームを SIO_delete 関数を使用して削除することはできません。

7.2.2 ストリームの動的な作成および削除

ストリームは、例 7-1 に示すように、実行時に SIO_create 関数を使用して作成することもできます。

例 7-1. SIO_create を使用してストリームを作成する方法

```
SIO_Handle SIO_create(name, mode, bufsize, attrs)
    String       name;
    Int          mode;
    Uns          bufsize;
    SIO_Attrs    *attrs;
```

SIO_create はストリームを作成し、SIO_Handle 型のハンドルを返します。SIO_create では、bufsize のサイズのバッファを指定して name に指定されているデバイスをオープンします。attrs にはオプション属性として、バッファの数、バッファ・メモリ・セグメント、ストリーミング・モデルなどを指定できます。mode パラメータは、ストリームが入力ストリーム（SIO_INPUT）か出力ストリーム（SIO_OUTPUT）かを指定するために使用します。

注：

name パラメータは、デバイス名の前にスラッシュ（/）を付けたものでなければなりません。たとえば sine という名前のデバイスの場合、name には、「/sine」を指定します。

ストリームをオープンするときにストリーミング・モデル（attrs→model）が SIO_STANDARD（デフォルト）にセットされている場合は、指定したサイズのバッファが割り当てられてそのストリーム用に使用されます。ストリーミング・モデルを SIO_ISSUERECLAIM にセットしてストリームをオープンした場合は、ストリームの作成者がすべての必要なバッファを供給するものと見なされるので、ストリーム・バッファの割り当ては行われません。

例 7-2 に示す SIO_delete は、関連するデバイスをクローズし、ストリーム・オブジェクトを解放します。SIO_STANDARD ストリーミング・モデルを使用してストリームがオープンされている場合は、ストリーム内に残っているすべてのバッファも解放されます。ユーザが保持するストリーム・バッファは、ユーザのコードによって明示的に解放する必要があります。

例 7-2.　ユーザが保持するストリーム・バッファを解放する方法

```
Int SIO_delete(stream)
    SIO_Handle     stream;
```

7.3 ストリーム入出力（I/O）— ストリームの読み取りと書き込み

DSP/BIOS には、データをストリーム処理するための 2 つのモデルがあります。標準モデル、および 発行（Issue）/ 再利用（Reclaim）モデルです。標準モデルはストリームを使用するための単純な手法を提供するだけですが、発行 / 再利用モデルはストリーム・オペレーションに対するさらに詳細な制御機能を提供します。

SIO_get と SIO_put は、例 7-3 に示すように標準ストリーミング・モデルを実装します。SIO_get は、データ・バッファを入力するために使用します。SIO_get は、ストリームとの間でバッファを交換します。bufp パラメータは、デバイスにバッファを渡し、別のバッファをアプリケーションに返すために使用します。SIO_get は、入力バッファのバイト数を返します。SIO_put 関数は、データ・バッファを出力し、そして SIO_get と同様にストリームとの間で物理バッファを交換します。SIO_put は、出力バッファ内のバイト数を取得します。

例 7-3. データ・バッファの入力方法と出力方法

```
Int SIO_get(stream, bufp)
    SIO_Handle      stream;
    Ptr             *bufp;

Int SIO_put(stream, bufp, nbytes)
    SIO_Handle    stream;
    Ptr           *bufp;
    Uns           nbytes;
```

注 :

bufp が指し示しているバッファがストリームとの間で交換されるので、バッファ・サイズ、メモリ・セグメント、アライメントは stream の属性に一致していなければなりません。

SIO_issue および SIO_reclaim は、例 7-4 に示すように、発行 / 再利用ストリーミング・モデルを実装する呼び出しです。SIO_issue は、ストリームにバッファを送ります。返されるバッファはなく、ストリームはブロックせずにタスクに制御を返します。arg は、DSP/BIOS では解釈されず、サービスとしてそのままストリーム・クライアントに提供されます。arg は、関連のバッファ・データと一緒に、各デバイスに渡されます。ストリーム・クライアントは、これをデバイス・ドライバとの通信手段として使用することができます。たとえば、arg を使用して、データを提供すべき正確な時刻を示すタイム・スタンプを出力デバイスに送ることができます。SIO_reclaim は、ストリームにバッファを返すよう要求します。

例 7-4.　発行 / 再利用ストリーミング・モデルを実装する方法

```
Int SIO_issue(stream, pbuf, nbytes, arg)
    SIO_Handle     stream;
    Ptr            pbuf;
    Uns            nbytes;
    Arg            arg;

Int SIO_reclaim(stream, bufp, parg)
    SIO_Handle     stream;
    Ptr            *bufp;
    Arg            *parg;
```

使用可能なバッファがない場合は、バッファが使用可能になるかまたはストリームのタイムアウト時間が経過するまで、ストリームはタスクをブロックします。

基本レベルでは、標準モデルと発行 / 再利用モデルとの間の最も顕著な相違点は、発行 / 再利用モデルでは、バッファ到着の通知（SIO_issue）とバッファが使用可能になるまでの待機（SIO_reclaim）が分離していることです。したがって、SIO_issue/SIO_reclaim のペアを使用すると、SIO_get または SIO_put を呼び出した場合と同じバッファ交換が行われます。

発行 / 再利用ストリーミング・モデルでは、ストリーム・クライアントが実行時に未処理バッファの数を制御できるので、高度な柔軟性を得ることができます。クライアントは SIO_issue を使用することにより、ブロックなしで 1 つのストリームに複数のバッファを送ることができます。バッファは、クライアントの要求に応じて、SIO_reclaim を呼び出すことにより返されます。したがってクライアントは、デバイスをどの程度までバッファするか、およびどの時点でブロックしてバッファを待つかを選択することができます。

さらに発行 / 再利用ストリーミング・モデルでは、クライアントのバッファは必ず発行された順序で返されるので、バッファ管理における確定性が高くなります。したがってクライアントは、どのようなソースからのメモリでもストリーミングに使用することができます。たとえば、ある DSP/BIOS タスクが 大きいバッファを受け取った場合、そのタスクはバッファを小さい断片に分けてストリームに渡すことができます。この操作に必要なのは、その大きいバッファ内でポインタを進めながら、個々の断片についてそれぞれ SIO_issue を呼び出すことだけです。このようなことが可能なのは、バッファの個々の断片が送信されたときと同じ順序で返されるという保証があるからです。

7.3.1　バッファ交換

DSP/BIOS のストリーミング・モデルの重要な役割の 1 つに、バッファ交換があります。小さいオーバーヘッドで効率的に入出力（I/O）オペレーションを行うために、DSP/BIOS は特定の入出力（I/O）オペレーションにおいて、データをある場所から別の場所へコピーすることを抑制します。コピーを行う代わりに、DSP/BIOS は、SIO_get、SIO_put、SIO_issue、SIO_reclaim を使用して、デバイスとの間でバッファ・ポインタをやりとりします。図 7-3 に、SIO_get の機能の概念を示します。

図 7-3.　SIO_get の働き

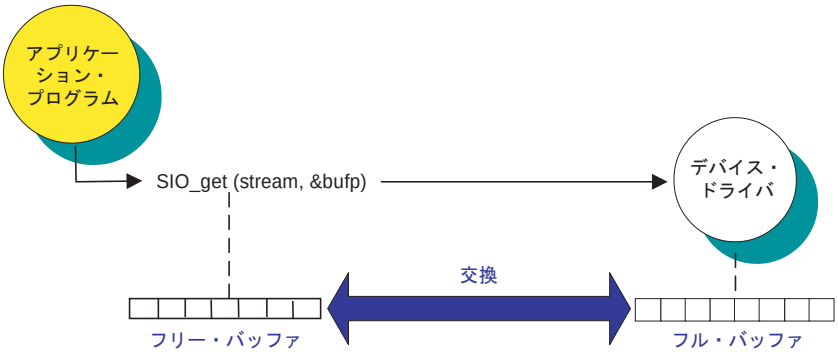


図 7-3 では、データが使用可能になると stream に関連付けられているデバイス・ドライバはバッファにデータを入れます。同時に、アプリケーション・プログラムは現行バッファの処理を進めます。アプリケーションが次のバッファを取得するために SIO_get を使用すると、入力デバイスにより満たされたバッファがすでに渡されているバッファとスワップされます。このときは、bufsize に相当するバイト数のデータがコピーされるのではなくバッファ・ポインタが交換されるだけなので、時間を大幅に節約できます。したがって、SIO_get のオーバーヘッドは、バッファ・サイズの影響を受けることがありません。

いずれの場合も、SIO_get により実際の物理バッファが変更されています。これに関する重要な考慮事項は、各オペレーションの後で、入出力（I/O）に使用するバッファに対するすべての参照を確実に更新する必要があるということです。この更新が行われないと、無効なバッファが参照されることになります。

SIO_put は、上記と同じポインタ交換を使用して出力ストリーム用のバッファを交換します。SIO_issue と SIO_reclaim は、どちらも 1 つの方向にしかデータを転送しません。したがって SIO_issue/SIO_reclaim のペアを使用すると、上記と同じバッファ・ポインタのスワップが行われることになります。

注：

1 つのストリームを、複数のタスクで同時に使用することはできません。つまり、アプリケーションの中で、1 つのストリームについて、SIO_get/SIO_put または SIO_issue/SIO_reclaim を同時に呼び出すことができるのは 1 つのタスクだけです。

7.3.2 例 - DGN デバイスから入力バッファを読み取る方法

例 7-5 のプログラムに、いくつかの基本 SIO 関数を示すとともに、ストリームからの読み取りを行う簡単な例を示します。DGN ソフトウェア生成プログラム・ドライバの詳細は、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「DGN」を参照してください。

例 7-5 の構成テンプレートは、DSP/BIOS 配布パッケージの siotest ディレクトリに収めてあります。ここでは、SIO ストリーム inputStream に対するデータ生成プログラムとして、sineWave という DGN デバイスが使用されています。タスク streamTask は、関数 doStreaming を呼び出して、inputStream から正弦データを読み取り、それをログ・バッファ trace に出力します。例 7-5 の出力は、図 7-4 に正弦波データとして示します。

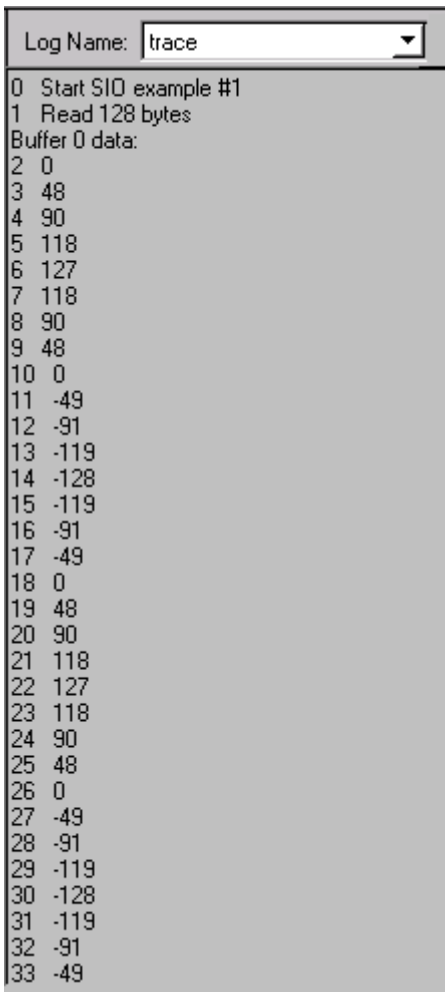
例 7-5. 基本 SIO 関数

```
/*
 *  ======== siotest1.c ========
 *  In this program a task reads data from a DGN sine device
 *  and prints the contents of the data buffers to a log buffer.
 *  The data exchange between the task and the device is done
 *  in a device independent fashion using the SIO module APIs.
 *
 *  The stream in this example follows the SIO_STANDARD streaming
 *  model and is created statically.
 *
 */

#include <std.h>

#include <log.h>
#include <sio.h>
#include <sys.h>
#include <tsk.h>

extern Int IDRAM1;     /* MEM segment ID defined by Conf tool */
extern LOG_Obj trace;  /* LOG object created with Conf tool */
extern SIO_Obj inputStream; /* SIO object created w Conf tool */
extern TSK_Obj streamTask;  /* pre-created task */

SIO_Handle input = &inputStream; /* SIO handle used below */

Void doStreaming(Uns nloops);    /* function for streamTask */

/*
 *  ======== main ========
 */
Void main()
{
    LOG_printf(&trace, "Start SIO example #1");
}

/*
```

例 7-5.　基本 SIO 関数（続き）

```
 *  ======== doStreaming ========
 *  This function is the body of the pre-created TSK thread
 *  streamTask.
 */
Void doStreaming(Uns nloops)
{
    Int i, j, nbytes;
    Int *buf;
    status = SIO_staticbuf(input, (Ptr *)&buf);    if (status !
= SYS_ok) {        SYS_abort(“could not acquire static frame:);
}

    for (i = 0; i < nloops; i++) {
        if ((nbytes = SIO_get(input, (Ptr *)&buf)) < 0) {
            SYS_abort("Error reading buffer %d", i);
        }

        LOG_printf(&trace, "Read %d bytes\nBuffer %d data:",
nbytes, i);
        for (j = 0; j < nbytes / sizeof(Int); j++) {
            LOG_printf(&trace, "%d", buf[j]);
        }
    }

    LOG_printf(&trace, "End SIO example #1");
}
```

図 7-4.　例 7-5 の出力トレース



7.3.3　例 – DGN デバイスの読み取りおよび書き込み方法

例 7-6 では、前の例に新しい SIO オペレーションが追加されています。出力ストリーム outputStream が、構成に追加されています。streamTask は、前の例と同様に DGN sine デバイスからバッファを読み取りますが、今度はその内容をログ・バッファに出力するのではなく、outputStream にデータ・バッファを送ります。outputStream ストリームは、printData いう名前の DGN ユーザ・デバイスにデータを送信します。printData デバイスは、受信したデータ・バッファを受け取り、DGN_print2log 関数を使用してログ・バッファの内容を表示します。ログ・バッファは、ユーザが構成時に指定したものです。

例 7-6.　例 7-5 へ出力ストリームを追加する方法

```
 ======== Portion of siotest2.c ========
/* SIO objects created with conf tool  */
extern far LOG_Obj trace;
extern far SIO_Obj inputStream;
extern far SIO_Obj outputStream;
extern far TSK_Obj streamTask; SIO_Handle input = &inputStream;
SIO_Handle output = &outputStream;
...

Void doStreaming(Uns nloops)
{
Void doStreaming(Arg nloops_arg)
{
    Int i, nbytes;
    Int *buf;
    Long nloops = (Long) nloops_arg;
    if ( SIO_staticbuf(input, (Ptr *)&buf) == 0) {
        SYS_abort("Error reading buffer ");
    }

    for (i = 0; i < nloops; i++) {
        if ((nbytes = SIO_get(input, (Ptr *)&buf)) < 0) {
            SYS_abort("Error reading buffer %d", (Arg)i);
        }
        if (SIO_put(output, (Ptr *)&buf, nbytes) < 0) {
            SYS_abort("Error writing buffer %d", (Arg)i);
        }
    }
    LOG_printf(&trace, "End SIO example #2");
}
/*  ======== DGN_print2log ========
 *  User function for the DGN user device printData. It takes as an argument
 *  the address of the LOG object where the data stream should be printed. */

Void DGN_print2log(Arg arg, Ptr addr, Uns nbytes)
{
    Int      i;
    Int      *buf;
    buf = (Int *)addr;

    for (i = 0; i < nbytes/sizeof(Int); i++) {
        LOG_printf((LOG_Obj *)arg, "%d", buf[i]);
    }
}
```

注：

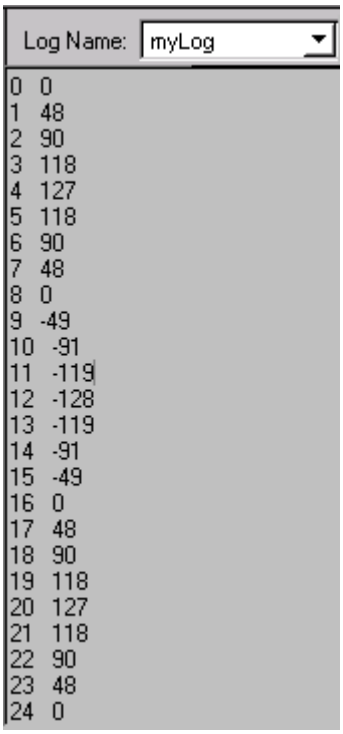
log_printf に渡す非ポインタ型の関数引数は、次のコード例に示すように、明示的に（Arg）に型キャストする必要があります。

```
LOG_printf(&trace, "Task %d Done", (Arg)id);
```

例 7-6 の完全なソース・コードと構成テンプレート（siotest2.c、siotest2.cdb、dgn_print.c）は、DSP/BIOS 製品配布パッケージの c:\ti\tutorial\target\siotest ディレクトリに収めてあります。DGN デバイスを静的に追加および構成する詳細については、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の「DGN」を参照してください。

この例の出力では、「myLog」ウィンドウに正弦波データが表示されます。

図 7-5.　例 7-6 の結果のウィンドウ



7.3.4　例 – 発行 / 再利用モデルを使用したストリーム入出力（I/O）

例 7-7 は、機能的には 例 7-6 と同じです。ただし、この例ではストリームは発行 / 再利用モデルを使用して作成され、ストリームへのデータの読み取りと書き込みを行う SIO オペレーションは SIO_issue と SIO_reclaim です。

このモデルでは、ストリームが動的に作成されたときには、最初はバッファは割り当てられていないので、アプリケーションで必要なバッファを割り当て、データ入出力（I/O）に使用するストリームにそれらのバッファを割り当てる必要があります。静的ストリームの場合は、構成で SIO オブジェクトに関する「Allocate Static Buffer(s)」チェックボックスにチェックマークを付けることにより、静的バッファを割り当てることができます。

例 7-7.　発行 / 再利用モデルの使用方法

```
 /*  ======== doIRstreaming ======== */
Void doIRstreaming(Uns nloops)
{
    Ptr    buf;
    Arg    arg;
    Int    i, nbytes;

    /* Prime the stream with a couple of buffers */
    buf = MEM_alloc(IDRAM1, SIO_bufsize(input), 0);
    if (buf == MEM_ILLEGAL) {
        SYS_abort("Memory allocation error");
    }
    /* Issue an empty buffer to the input stream */
    if (SIO_issue(input, buf, SIO_bufsize(input), NULL) < 0) {
        SYS_abort("Error issuing buffer %d", i);
    }

    buf = MEM_alloc(IDRAM1, SIO_bufsize(input), 0);
    if (buf == MEM_ILLEGAL) {
        SYS_abort("Memory allocation error");
    }

    for (i = 0; i < nloops; i++) {
        /* Issue an empty buffer to the input stream */
        if (SIO_issue(input, buf, SIO_bufsize(input), NULL) < 0) {
            SYS_abort("Error issuing buffer %d", i);
        }
        /* Reclaim full buffer from the input stream */
        if ((nbytes = SIO_reclaim(input, &buf, &arg)) < 0) {
            SYS_abort("Error reclaiming buffer %d", i);
        }
        /* Issue full buffer to the output stream */
        if (SIO_issue(output, buf, nbytes, NULL) < 0) {
            SYS_abort("Error issuing buffer %d", i);
        }
        /* Reclaim empty buffer from the output stream to be reused */
        if (SIO_reclaim(output, &buf, &arg) < 0) {
            SYS_abort("Error reclaiming buffer %d", i);
        }
    }
    /* Reclaim and delete the buffers used */
    MEM_free(IDRAM1, buf, SIO_bufsize(input));
    if ((nbytes = SIO_reclaim(input, &buf, &arg)) < 0) {
        SYS_abort("Error reclaiming buffer %d", i);
    }
    if (SIO_issue(output, buf, nbytes, NULL) < 0) {
        SYS_abort("Error issuing buffer %d", i);
        }
    if (SIO_reclaim(output, &buf, &arg) < 0) {
        SYS_abort("Error reclaiming buffer %d", i);
    }

    MEM_free(IDRAM1, buf, SIO_bufsize(input));
}
```

この例の完全なソース・コードと構成テンプレートは、DSP/BIOS 製品配布パッケージの C:\ti\tutorial\target\siotest フォルダに収めてあります（target はご使用のプラットフォームです）。例 7-7 の出力は、例 7-5 の場合と同じです。

7.4 スタック可能デバイス

SIO モジュールの機能は、DSP/BIOS におけるデバイス独立性を促進するための重要な役割を果たします。つまり論理デバイスを使用することにより、特定のデバイスを指定するための細かい作業からアプリケーション・プログラムを解放できます。たとえば /dac は、特定の DAC ハードウェアを表さない論理デバイス名です。このデバイス命名方式により、デバイス独立入出力（I/O）に関して DSP/BIOS に固有の新たな特質が加わります。つまり、単一の名前でデバイスのスタックを表すことができるという能力です。

注：

データ・ストリーミング・デバイスまたはメッセージ受け渡しデバイスを次々にスタッキングして仮想入出力（I/O）デバイスを作成することにより、さらにアプリケーションをその基礎にあるシステム・ハードウェアから分離することができます。

一例として、あるプログラムで実装されているアルゴリズムが一対の A/D-D/A コンバータを使用して固定小数点データのストリームの入出力を行う場合を考えてみてください。ただし、この A/D-D/A デバイスが取得できるのは 14 個の最上位ビットのデータだけなので、入力データを拡張したい場合は残りの 2 ビットは 0 でなければなりません。

この場合、アルゴリズムの要件を満たすためのデータ変換や、バッファリング用の余分なコードでプログラムが混乱するのを避けるために、例 7-8 に示すように、一対の仮想デバイスをオープンして、その基礎にある複数の実デバイスが生成し消費するデータに対して一連の変換を暗黙的に行うことができます。

例 7-8. 一対の仮想デバイスをオープンする方法

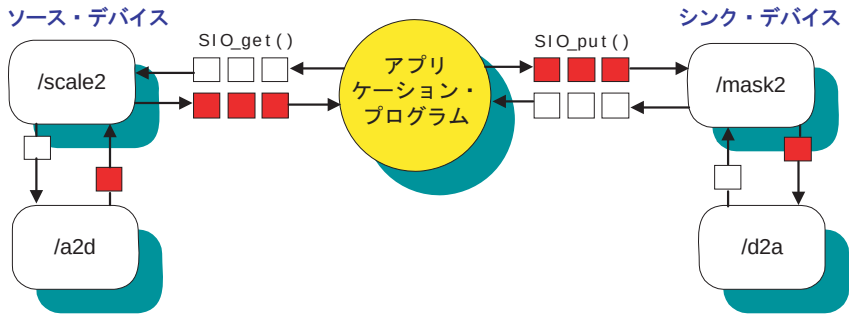
```
SIO_Handle input;
SIO_Handle output;
Ptr        buf;
Int        n;

buf = MEM_alloc(0, MAXSIZE, 0);

input = SIO_create("/scale2/a2d", SIO_INPUT, MAXSIZE, NULL);
output = SIO_create("/mask2/d2a", SIO_OUTPUT, MAXSIZE, NULL);

while (n = SIO_get(input, &buf)) {

    `apply algorithm to contents of buf`

    SIO_put(output, &buf, n);
}

SIO_delete(input);
SIO_delete(output);
```

例 7-8 では、仮想入力デバイス /scale2/a2d は実際には 2 つのデバイスのスタックから構成されており、各デバイスは構成ファイル内で指定されているデバイス名のプレフィックスに従って命名されています。

- **/scale2** は、基礎デバイス（/a2d）が生成する固定小数点データ・ストリームを、位取りされた固定小数点値のストリームに変換するデバイスを表します。
- **/a2d** は、A/D-D/A デバイス・ドライバが管理するデバイスで、A/D コンバータから固定小数点入力のストリームを生成します。

仮想出力デバイス /mask2/d2a も 2 つのデバイスのスタックを表します。図 7-6 は、アプリケーション・プログラムが SIO データ・ストリーミング関数を呼び出したときの、これらの仮想ソース・デバイスとシンク・デバイスを介した空フレームとフル・フレームの流れを示しています。

図 7-6.　空フレームとフル・フレームの流れ



7.4.1　例 – SIO_create とスタッキング・デバイス

例 7-9 では、sourceTask および sinkTask という 2 つのタスクがパイプ・デバイスを介してデータを交換します。

sourceTask は、ライタ・タスクで DGN sine デバイスに接続している入力ストリームからデータを受け取り、そのデータを DPI パイプ・デバイスに接続している出力ストリームに直接渡します。入力ストリームにも、DGN sine デバイスの上にスタッキング・デバイス（scale）があります。sine から送られたデータ・ストリームは、まず scale デバイスにより処理され（各データ・ポイントに定数整数値が掛けられる）、それを sourceTask が受け取ります。

sinkTask は、リーダ・タスクで sourceTask が入力ストリームとして DPI pipe デバイスに送ったデータを読み取り、そのデータを出力ストリームを通して DGN printData に直接渡します。

例 7-9 で使用するデバイスは、静的に構成されています。例 7-9 の完全なソース・コードと構成テンプレート（siotest5.c、siotest5.cdb、dgn_print.c）は、DSP/BIOS 製品配布パッケージの c:\ti\tutorial\target\siotest ディレクトリに収めてあります。デバイス sineWave および printDat は DGN デバイス、pip0 は DPI デバイス、scale は DTR スタッキング・デバイスです。DPI、DGN、および DTR デバイスの追加方法と構成方法は、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の DPI、DGN、および DTR ドライバについての説明を参照してください。

例 7-9 のストリームも構成に追加されています。sourceTask タスクの入力ストリームは inStreamSrc です。

スタッキング・デバイスを使用する SIO ストリームを構成するときは、まず「Device Control Parameter」プロパティに構成済みの端末デバイスを入力する必要があります。端末デバイスの名前の前にはスラッシュ（/）を付けなければなりません。例では、/sineWave を使用しています（sineWave は構成済みの DGN 端末デバイスの名前です）。次に、「Device」プロパティのドロップダウン・リストからスタッキング・デバイス（scale）を選択します。構成時に、「Device Control Parameter」に端末デバイスを入力するまでは「Device」でスタッキング・デバイスを選択できません。例 7-9 では、その他の SIO ストリームとして、outStreamSrc（sourceTask 用の出力ストリーム）、inStreamSink（sinkTask 用の入力ストリーム）、および outStreamSink（sinkTask 用の出力ストリーム）が作成されています。これらのストリームが使用するデバイスは、端末デバイス pip0 および printData です。

例 7-9.　パイプ・デバイスを使用したデータ交換

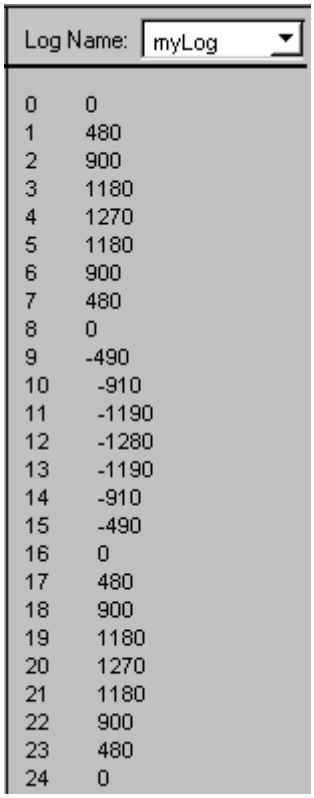
```
/*
 *  ======== siotest5.c ========
 *  In this program two tasks are created that exchange data
 *  through a pipe device. The source task reads sine wave data
 *  from a DGN device through a DTR device stacked on the sine
 *  device, and then writes it to a pipe device. The sink task
 *  reads the data from the pipe device and writes it to the
 *  printData DGN device. The data exchange between the tasks
 *  and the devices is done in a device independent fashion
 *  using the SIO module APIs.
 *
 *  The streams in this example follow the SIO_STANDARD streaming
 *  model and are created statically.
 */

#include <std.h>

#include <dtr.h>
#include <log.h>
#include <mem.h>
#include <sio.h>
#include <sys.h>
#include <tsk.h>

#define BUFSIZE 128

#ifdef _62_
#define SegId IDRAM
extern Int IDRAM;    /* MEM segment ID defined with conf tool */
#endif

#ifdef _54_
#define SegId IDATA
extern Int IDATA;    /* MEM segment ID defined with conf tool */
#endif

#ifdef _55_
#define SegId DATA
extern Int DATA;     /* MEM segment ID defined with conf tool */
#endif

extern LOG_Obj trace;       /* LOG object created with conf tool */
extern TSK_Obj sourceTask;  /* TSK thread objects created via conf tool */
extern TSK_Obj sinkTask;
extern SIO_Obj inStreamSrc; /* SIO streams created via conf tool */
extern SIO_Obj outStreamSrc;
extern SIO_Obj inStreamSink;
extern SIO_Obj outStreamSink;

/* Parameters for the stacking device  "/scale" */
DTR_Params DTR_PRMS = {
    20,   /* Scaling factor */
    NULL,
    NULL
};

Void source(Uns nloops);    /* function body for sourceTask above */
Void sink(Uns nloops);      /* function body for sinkTask above */

static Void doStreaming(SIO_Handle input, SIO_Handle output, Uns nloops);

/*
```

例 7-9.　パイプ・デバイスを使用したデータ交換（続き）

```
 *  ======== main ========
 */
Void main()
{
    LOG_printf(&trace, "Start SIO example #5");
}

/*
 *  ======== source ========
 *  This function forms the body of the sourceTask TSK thread.
 */
Void source(Uns nloops)
{
    SIO_Handle input = &inStreamSrc;
    SIO_Handle output = &outStreamSrc;

    /* Do I/O */
    doStreaming(input, output, nloops);
}

/*
 *  ======== sink ========
 *  This function forms the body of the sinkTask TSK thread.
 */
Void sink(Uns nloops)
{
    SIO_Handle input = &inStreamSink;
    SIO_Handle output = &outStreamSink;

    /* Do I/O */
    doStreaming(input, output, nloops);

    LOG_printf(&trace, "End SIO example #5");
}

/*
 *  ======== doStreaming ========
 *  I/O function for the sink and source tasks.
 */
static Void doStreaming(SIO_Handle input, SIO_Handle output, Uns nloops)
{
    Ptr    buf;
    Int    i, nbytes;

    if (SIO_staticbuf(input, &buf) == 0){
        SYS_abort("Eror reading buffer %d", i);
    }
    for (i = 0; i < nloops; i++) {
         if ((nbytes = SIO_get (input, &buf)) <0) {
             SYS_abort ("Error reading buffer %d", i);
        }
        if (SIO_put (output, &buf, nbytes) <0) {
             SYS_abort ("Error writing buffer %d", i);
        }
    }
}
```

例 7-9 の出力では、例 7-7 に示す「myLog」ウィンドウに正弦波データが表示されます。

図 7-7.　例 7-9 の正弦波出力



sioTest5.c を編集して DTR_PRMS の位取り係数を変更し、実行ファイルをリビルドすると、myLog への出力の相違を見ることができます。

製品配布パッケージには、例 7-9 のバージョンの 1 つとして、実行時に SIO_create を呼び出すことにより動的にストリームを作成する場合の例（siotest4.c、siotest4.cdb）も収めてあります。

7.5 ストリームの制御

一般に、物理デバイスに望みどおりの動作をさせるためには、1 つまたは複数の特殊な制御信号が必要です。SIO_ctrl を使用すると、デバイスと通信してコマンドと引数を渡すことができます。各デバイスはそれぞれ専用のコマンドだけを受け入れるので、各デバイス固有の資料を参照する必要があります。一般的な呼び出しフォーマットを例 7-10 に示します。

例 7-10. SIO_ctrl を使用してデバイスと通信する方法

```
Int SIO_ctrl(stream, cmd, arg)
    SIO_Handle stream;
    Uns        cmd;
    Ptr        arg;
```

stream に関連付けられているデバイスに、デバイス固有の cmd が表すコマンドが渡されます。コマンドの引数を指す汎用ポインタもこのデバイスに渡されます。次に、デバイス・ドライバの一部として組み込まれている実際の制御関数がコマンドと引数を解釈し、それに基づいて処理を実行します。

AD コンバータ・デバイス /a2d が、サンプリング・レートを変更するための制御オペレーションを行うものとします。例 7-11 に示すように、サンプリング・レートは 12 kHz に変更されます。

例 7-11. サンプリング・レートの変更

```
SIO_Handle     stream;

stream = SIO_create("/a2d", ...);

SIO_ctrl(stream, DAC_RATE, 12000);
```

状況によっては、バッファド入出力（I/O）を行っている入出力（I/O）デバイスとの同期をとることができます。デバイスとの同期をとる方法には、SIO_idle と SIO_flush の 2 つがあります。どちらの関数もデバイスをアイドル状態にします。デバイスをアイドル状態にすると、すべてのバッファがデバイスの初期作成時に入っていたキューに戻されます。つまりデバイスは初期状態に戻り、ストリーミングは停止します。

入力ストリームの場合、この 2 つの関数は同じ結果をもたらします。つまり、読み取られていない入力はすべて失われます。出力ストリームの場合、SIO_idle はバッファにあるすべてのデータをデバイスに書き込むまでブロックします。しかし SIO_flush はまだデバイスに書き込まれていないデータをすべて破棄します。例 7-12 に示すように、SIO_flush はブロックしません。

例 7-12. デバイスとの同期をとる方法

```
Void SIO_idle(stream);
    SIO_Handle     stream;
Void SIO_flush(stream);
    SIO_Handle     stream;
```

アイドル・ストリームは、基礎デバイスとの入出力（I/O）処理を行いません。したがって、SIO_idle または SIO_flush を呼び出すことにより、さらに後続の入力や出力が必要になるまでストリームをオフにできます。

7.6 複数ストリーム間の選択

SIO_select 関数を使用すると、一組の SIO ストリームのうちの 1 つまたは複数のストリームに対する入出力（I/O）オペレーションが可能になるまで、1 つの DSP/BIOS タスクをブロックせずに待つようにすることができます。このメカニズムは、たとえば次のようなアプリケーションで使用すると便利です。

- **非ブロッキング入出力（I/O）。**データを低速デバイス（たとえばディスク・ファイル）にストリーミングするリアルタイム・タスクは、SIO_put がブロックしないようにする必要があります。
- **マルチタスキング。**事実上すべてのマルチタスキング・アプリケーションに、複数のソースからのデータをルーティングするデーモン・タスクがあります。SIO_select のメカニズムを使用すると、1 つのタスクでこれらすべてのソースを扱えるようにすることができます。

SIO_select を呼び出すには、ストリームの配列、配列の長さ、およびタイムアウト値を指定します。SIO_select は、ストリームのいずれかがレディ状態になるか、またはタイムアウト時間が経過するまでブロックします（ただし timeout が 0 以外の場合）。どちらの場合も、SIO_select が返すマスクは、例 7-13 に示すように、どのデバイスがサービス対象としてレディ状態になっているかを示します（ビット j が 1 であれば、streamtab[j] がレディ状態にあることを示します）。

例 7-13. ストリームがレディ状態にあることを示す方法

```
Uns SIO_select(streamtab, nstreams, timeout)
    SIO_Handle     streamtab[];   /* stream table */
    Uns            nstreams;      /* number of streams */
    Uns          timeout;      /* return after this many */
                                  /* system clock ticks */
```

7.6.1 プログラミング例

例 7-14 では、入出力（I/O）オペレーションがブロックするかどうかを検出するために 2 つのストリームをポーリングしています。

例 7-14.　2 つのストリームのポーリング

```
SIO_Handle     stream0;
SIO_Handle     stream1;
SIO_Handle     streamtab[2];
Uns            mask;

...

streamtab[0] = stream0;
streamtab[1] = stream1;

while ((mask = SIO_select(streamtab, 2, 0)) == 0) {

    `I/O would block, do something else`

}

if (mask & 0x1) {
    `service stream0`
}
if (mask & 0x2) {
    `service stream1`
}
```

7.7 複数クライアントへのデータのストリーミング

マルチプロセッシング・システムに共通する問題点の 1 つに、システム内の複数タスクへの同一データ・バッファの同時伝送があります。このようなマルチキャスト伝送、つまりデータの分散は、DSP/BIOS の SIO ストリームを使用することにより簡単に行うことができます。単一のプロセッサが 4 つのクライアント・プロセッサにデータを送信する場合を考えてみましょう。

この状況においてプロセッサ間のデータをストリーミングするのは、1 つのバッファのデータを 1 つまたは複数のクライアントに送られなければならないという点で、獲得デバイスとの間でデータをストリーミングするのとは異なります。データ入出力（I/O）には、DSP/BIOS の SIO 関数 SIO_get/SIO_put が使用されます。

SIO_put は、すでにデバイス・レベルにあるバッファとアプリケーション・バッファの間で自動的にバッファ交換を行います。その結果、バッファは入出力（I/O）用としてエンキューされ、入出力（I/O）は割り込みレベルで非同期的に発生するため、ユーザはバッファを制御できなくなります。したがって、複数のクライアントにデータを送信するためには、ユーザはデータをコピーしなければなりません。例 7-15 に、これを示します。

例 7-15. SIO_put を使用して複数のクライアントにデータを送信する方法

```
SIO_put(inStream, (Ptr)&bufA, npoints);

`fill bufA with data`
for (`all data points`) {
    bufB[i] = bufC[i] = bufD[i] ... = bufA[i];
}
SIO_put(outStreamA, (Ptr)&bufA, npoints);
SIO_put(outStreamB, (Ptr)&bufB, npoints);
SIO_put(outStreamC, (Ptr)&bufC, npoints);
SIO_put(outStreamD, (Ptr)&bufD, npoints);
```

データをコピーするためには各ストリームがそれぞれバッファを必要とするので、CPU サイクルが余分に消費され、多くのメモリが必要になります。二重バッファリングを行う場合には、例 7-15 では 8 つのバッファ（各システムについて 2 つずつ）が必要です。

例 7-16 に、この状況で SIO_issue と SIO_reclaim を使用した場合の利点を示します。この例ではアプリケーションはコピーを行わず、使用されるバッファは 2 つだけです。1 回の呼び出しで SIO_issue が行うのは、bufA が指すバッファをブロックすることなく outStream の todevice キューにエンキューすることだけです。この方法ではコピーもブロックも行われないので、データのバッファを伝送可能な状態にしてから、バッファをすべてのクライアントに送信できるようになるまでの時間が大幅に短縮されます。出力デバイスからバッファを削除するには、対応する SIO_reclaim 関数を呼び出す必要があります。

例 7-16. SIO_issue/SIO_reclaim を使用して複数のクライアントにデータを送信する方法

```
SIO_issue(outStreamA, (Ptr)bufA, npoints, NULL);
SIO_issue(outStreamB, (Ptr)bufA, npoints, NULL);
SIO_issue(outStreamC, (Ptr)bufA, npoints, NULL);
SIO_issue(outStreamD, (Ptr)bufA, npoints, NULL);

SIO_reclaim(outStreamA, (Ptr)&bufA, NULL);
SIO_reclaim(outStreamB, (Ptr)&bufA, NULL);
SIO_reclaim(outStreamC, (Ptr)&bufA, NULL);
SIO_reclaim(outStreamD, (Ptr)&bufA, NULL, SYS_FOREVER);
```

注 :

バッファ内のデータを変更するデバイス・ドライバの場合、バッファは同時に複数のデバイスに送られるので、SIO_issue を使用して同じバッファを複数のデバイスに送ることはできません。たとえば、パック・データをアンパック・データに変換するスタッキング・デバイスは、他のデバイスがバッファを出力しているときに同時にバッファを変更します。

SIO_issue インターフェイスは、すべての通信ドライバが同じデータ・バッファにアクセスできるようにするための手段を提供します。各通信デバイス・ドライバは、一般に DMA 転送を使用してこのデータ・バッファを並行して転送します。すべてのストリームからバッファが使用可能になるまでは、プログラムは 4 つの SIO_reclaim から戻りません。

つまり SIO_issue/SIO_reclaim 関数は、複数のストリームへの同時データ伝送を行うための最も効率的な手段を提供します。ただし、これは双方向オペレーションではありません。SIO_issue/SIO_reclaim モデルは、出力に関しては分散の問題を極めて効果的に解決しますが、逆に複数のデータ・ソースを入力用として 1 つのバッファにまとめることはできません。

7.8 ターゲットとホスト間のデータのストリーミング

ホスト・チャネル・オブジェクト（HST オブジェクト）を作成して、アプリケーションがターゲットとホスト上のファイルの間でデータを転送するよう構成することができます。そのためには、DSP/BIOS 解析ツール でこれらのチャネルをホスト・ファイルにバインドし、そして起動します。

DSP/BIOS に組み込まれているホスト入出力（I/O）モジュール（HST）を使用すると、ホスト・コンピュータとターゲット・プログラムの間で簡単にデータを転送できます。各ホスト・チャネルは、内部的には SIO ストリーム・オブジェクトを使用して実現されます。ホスト・チャネルを使用するために、プログラムは HST_getstream を呼び出して対応するストリーム・ハンドルを取得した後、ストリーム上で SIO 呼び出しを使用してデータを転送します。

ホスト・チャネル（HST オブジェクト）を入力用または出力用として構成できます。入力チャネルはホストからターゲットにデータを転送し、出力チャネルはターゲットからホストにデータを転送します。

7.9 デバイス・ドライバ・テンプレート

デバイス・ドライバはハードウェアと直接対話するので、デバイス・ドライバに関する低レベルの詳細事項はドライバごとに大きく変化します。ただし、すべてのデバイス・ドライバは、SIO に同じインターフェイスを提供するものでなければなりません。次の節では、Dxx という名前のドライバ・テンプレートの例を示します。このテンプレートには、主に高レベル・オペレーション用の C コード、および低レベル・オペレーション用の疑似コードが含まれています。デバイス・ドライバは、Dxx 関数が行う標準動作に従うものでなければなりません。

1 つまたは複数の実際のドライバのソース・コードに合わせて、Dxx ドライバ・テンプレートの説明を読んでください。また、ご使用のプラットフォームに対応した『TMS320 DSP/BIOS API Reference Guide』の Dxx 関数についての説明も役立ちます。なお、この場合の xx は 2 文字の組み合わせです。カスタム・ドライバ、および DSP/BIOS 提供のドライバを含めたデバイス・ドライバの構成方法の詳細は、該当のデバイス・ドライバに関する資料の説明を参照してください。

7.9.1 一般的なファイル編成

通常、デバイス・ドライバは複数のファイルに分かれています。次に例を示します。

- dxx.h − Dxx ヘッダ・ファイル
- dxx.c − Dxx 関数
- dxx_asm.s## −（オプション）アセンブリ言語関数

ほとんどのデバイス・ドライバ・コードは C で記述することができます。次の Dxx の説明では、アセンブリ言語は使用しません。ただし、割り込みサービス・ルーチンは効率化のためにアセンブリ言語で記述することが多く、また一部のハードウェア制御関数もアセンブリ言語で書く必要があります。

この時点で、DGN などのソフトウェア・デバイス・ドライバのレイアウトについてよく理解しておくことをお勧めします。特に、次の点に注意する必要があります。

- 通常、ヘッダ・ファイル dxx.h には、デバイス固有の定義のほかに、例 7-17 に示す文を含める必要があります。

例 7-17.　dxx.h ヘッダ・ファイル内に必要な文

```
/*
 *  ======== dxx.h ========
 */

#include <dev.h>
extern DEV_Fxns    Dxx_FXNS;
/*
 * ======== Dxx_Params ========
 */
typedef struct {

    `device parameters go here`

} Dxx_Params;
```

- ❑ Dxx_Params などのデバイス・パラメータは、構成時にデバイス・オブジェクトのプロパティとして指定します。

デバイス関数に必要なテーブルは、dxx.c に含めます。SIO モジュールは、このテーブルを使用して特定のデバイス・ドライバ関数を呼び出します。たとえば、SIO_put は、このテーブルを使用して、Dxx_issue/Dxx_reclaim を見つけて呼び出します。例 7-18 に、このテーブルを示します。

例 7-18.　デバイス関数のテーブル

```
DEV_Fxns Dxx_FXNS = {
    Dxx_close,
    Dxx_ctrl,
    Dxx_idle,
    Dxx_issue,
    Dxx_open,
    Dxx_ready,
    Dxx_reclaim
```

7.10 DEV 構造体のストリーミング

DEV_Fxns 構造体には、例 7-19 に示すように、汎用入出力（I/O）オペレーションに対応する内部ドライバ関数を指すポインタが含まれています。

例 7-19. DEV_Fxns 構造体

```
typedef struct DEV_Fxns {
    Int     (*close)(DEV_Handle);
    Int     (*ctrl)(DEV_Handle, Uns, Arg);
    Int     (*idle)(DEV_Handle, Bool);
    Int     (*issue(DEV_Handle);
    Int     (*open)(DEV_Handle, String);
    Bool    (*ready)(DEV_Handle, SEM_Handle);
    Int     (*reclaim)(DEV_Handle);
} DEV_Fxns;
```

デバイス・フレームは、DEV_Frame 型の構造体で、SIO およびデバイス・ドライバはこの構造体を使用してストリーム・バッファをエンキューまたはデキューします。device→todevice キューおよび device→fromdevice キューは、この型の要素を含んでいます（例 7-20）。

例 7-20. DEV_Frame 構造体

```
typedef struct DEV_Frame { /* frame object */
    QUE_Elem   link;    /* queue link */
    Ptr        addr;    /* buffer address */
    Uns        size;    /* buffer size */
    Arg        misc;    /* reserved for driver */
    Arg        arg;     /* user argument */
    Uns        cmd;     /* mini-driver command */
    Int        status;  /* status of command */
} DEV_Frame;
```

例 7-20 には次のパラメータがあります。

- *link* は、フレームをエンキューまたはデキューするために、QUE_put および QUE_get により使用されます。
- *addr* は、ストリーム・バッファのアドレスを含みます。
- *addr* は、ストリーム・バッファの論理サイズを含みます。論理サイズは、物理バッファ・サイズより小さくても構いません。
- *misc* は、デバイスが使用するために予約されているフィールドです。
- *arg* は、ユーザが特定のデータ・フレームに情報を関連付けるために使用できるフィールドです。このフィールドは、デバイスにより予約されます。
- *cmd* は、IOM モデルを使用するミニドライバとともに使用するためのコマンド・コードで『DSP/BIOS Driver Developer's Guide』（文献番号 SPRU616）で説明しています。コマンド・コードは、ミニドライバに、どのアクションを取るかを伝えます。
- *status* は、コールバック関数を呼び出す前に、IOM ミニドライバによりセットされるフィールドです。

デバイス・ドライバ関数には、最初または唯一のパラメータとして DEV_Handle を指定します。その上で追加のパラメータを指定します。DEV_Handle は DEV_Obj を指すポインタです。これは SIO_create により作成されて初期化され、Dxx_open に渡されてさらに初期化されます。DEV_Obj について特に重要なのは、SIO およびデバイスがバッファの交換に使用するバッファ・キューを指すポインタが含まれているという点です。すべてのドライバ関数には、最初のパラメータとして DEV_Handle を指定します。

例 7-21.　DEV_Handle 構造体

```
typedef DEV_Obj *DEV_Handle;

typedef struct DEV_Obj { /* device object */
   QUE_Handle   todevice;   /* downstream frames here */
   QUE_Handle   fromdevice; /* upstream frames here */
   Uns          bufsize;    /* buffer size */
   Uns          nbufs;      /* number of buffers */
   Int          segid;      /* buffer segment ID */
   Int          mode;       /* DEV_INPUT/DEV_OUTPUT */
   LgInt        devid;      /* device ID */
   Ptr          params;     /* device parameters */
   Ptr          object;     /* ptr to dev instance obj */
   DEV_Fxns     fxns;       /* driver functions */
   Uns          timeout;    /* SIO_reclaim timeout value */
   Uns          align;      /* buffer alignment */
   DEV_Callback *callback;  /* pointer to callback */
} DEV_Obj;
```

例 7-21 には次のパラメータがあります。

- *todevice* は、DEV_Frame フレームをデバイスに送るために使用されます。SIO_STANDARD（DEV_STANDARD）ストリーミング・モデルでは、SIO_put がフル・フレームをこのキューに入れ、SIO_get が空フレームをここに入れます。SIO_ISSUERECLAIM（DEV_ISSUERECLAIM）ストリーミング・モデルでは、SIO_issue がフレームをこのキューに入れます。
- *fromdevice* は、デバイスから DEV_Frame フレームを転送するために使用されます。SIO_STANDARD（DEV_STANDARD）ストリーミング・モデルでは、SIO_put がこのキューから空フレームを取得し、SIO_get がここからフル・フレームを取得します。SIO_ISSUERECLAIM（DEV_ISSUERECLAIM）ストリーミング・モデルでは、SIO_reclaim がこのキューからフレームを検索します。
- *bufsize* は、デバイス・キュー内のバッファの物理サイズを指定します。
- *nbufs* は、SIO_STANDARD ストリーミング・モデルでこのデバイス用に割り当てられているバッファの数か、SIO_ISSUERECLAIM ストリーミング・モデルの中の未解決バッファの最大数を指定します。
- *segid* は、デバイス・バッファがどのセグメントから割り当てられたかを示します（SIO_STANDARD）。

- *mode* は、そのデバイスが入力デバイス（DEV_INPUT）か出力デバイス（DEV_OUTPUT）かを指定します。
- *devid* はデバイス ID です。
- *params* は、任意のデバイス固有パラメータを指す汎用ポインタです。デバイスによっては、ここに示される追加のパラメータを必要とする場合があります。
- *object* は、デバイス・オブジェクトを指すポインタです。ほとんどのデバイスは、連続するデバイス・オペレーションで参照されるオブジェクトを作成します。
- *fxns* は、ドライバの関数を含む DEV_Fxns 構造体です。通常、この構造体は Dxx_FXNS のコピーですが、ドライバは Dxx_open でこれらの機能を動的に変更できます。
- *timeout* は、SIO_reclaim が 入出力（I/O）が完了するまでに待つシステム・ティックの数を指定します。
- *align* は、バッファ・アライメントを指定します。
- *callback* は、チャネル固有のコールバック構造体へのポインタを指定します。DEV_Callback 構造体は、コールバック関数と 2 つの関数引数を含みます。コールバック関数は、通常 SWI_andnHook または、SWI をポストするよく似た関数です。コールバックは、発行 / 再利用モデルと共にのみ使用できます。このコールバックにより、SIO オブジェクトは、SWI スレッドと共に使用できます。

ドライバの関数により変更するのは、object フィールドと fxns フィールドだけにしてください。これらのフィールドは、実質的には Dxx_open の出力パラメータです。

7.11 デバイス・ドライバの初期化

7.10 節「DEV 構造体のストリーミング」（7-30 ページ）で述べたように、ドライバ関数テーブル Dxx_FXNS は dxx.c の中で初期化されます。

さらに、Dxx_init により追加の初期化が行われます。Dxx モジュールは、他のアプリケーション・レベルのモジュールが初期化されるときに初期化されます。通常、Dxx_init は、例 7-22 に示すようにハードウェア初期化ルーチンを呼び出し、静的ドライバ構造体を初期化します。

例 7-22. Dxx_init による初期化

```
/*
 *  ======== Dxx_init ========
 */

Void Dxx_init()
{
    `Perform hardware initialization`
}
```

Dxx_init は、DSP/BIOS の構成および初期化標準との一貫性を維持するために必要なものですが、Dxx_init の内部オペレーションについては実質的な DSP/BIOS の要件は何もありません。ハードウェア初期化については事実上標準はなく、システムによっては、Dxx 内の別のところ（Dxx_open など）で所定のハードウェア・セットアップ・オペレーションを行う方が妥当な場合があります。したがって、一部のシステムでは Dxx_init は単なる空の関数の場合もあります。

7.12 デバイスのオープン

Dxx_open は、例 7-23 に示すように、Dxx デバイスをオープンし、そのステータスを返します。

例 7-23. Dxx_open を使用してデバイスをオープンする方法

```
status = Dxx_open(device, name);
```

SIO_create は、例 7-24 に示すように、Dxx_open を呼び出して Dxx デバイスをオープンします。

例 7-24. 入力終端デバイスをオープンする方法

```
input = SIO_create("/adc16", SIO_INPUT, BUFSIZE, NULL)
```

次のステップは、入力終端デバイスをオープンする手順を示しています。

1) DEV_devtab デバイス・テーブル内で、/adc16 というプレフィックスをもつ文字列を検索します。関連する DEV_Device 構造体には、ドライバ関数、デバイス ID、およびデバイス・パラメータが含まれています。

2) DEV_Obj デバイス・オブジェクトを割り当てます。

3) パラメータおよび SIO_create に渡される SIO_Attrs に基づいて、DEV_Obj の bufsize、nbufs、segid などのフィールドを指定します。

4) todevice キューと fromdevice キューを作成します。

5) DEV_STANDARD ストリーミング・モデル用にオープンした場合は、サイズが BUFSIZE の attrs.nbufs 個のバッファを割り当てて、todevice キューに入れます。

6) 次の構文を使用して、新しい DEV_Obj を指すポインタと残りの名前の文字列を指定して Dxx_open を呼び出します。

```
status - Dxx_open (device, "16")
```

7) device が指している DEV_Obj 内のフィールドを有効にします。

8) 追加のパラメータ（たとえば、16 kHz）用の文字列を構文解析します。

9) デバイス固有のオブジェクトを割り当てたり初期化したりします。

10) デバイス固有のオブジェクトを device→object に指定します。

例 7-25 に、Dxx_open への引数を示します。

例 7-25. Dxx_open への引数

```
DEV_Handle device;    /* driver handle */
String     name;      /* device name */
```

device パラメータは、DEV_Obj 型のオブジェクトを指し示します。このオブジェクトのフィールドは、SIO_create により初期化されています。name は、SIO_create が DEV_match を使用してデバイス名を照合した後の残りの文字列です。

SIO_create は、例 7-26 に示すパラメータを使用して呼び出されます。

例 7-26. SIO_create のパラメータ

```
stream = SIO_create(name, mode, bufsize, attrs);
```

通常、SIO_create に渡される name パラメータは、デバイスを示す文字列に、そのデバイスの動作モードを示すサフィックスを付加したものです。AD コンバータのベース名は /adc であり、サンプリング周波数は 16 などのタグ（16 kHz の場合）で示されます。この場合、SIO_create に渡される完全名は /adc16 となります。

SIO_create は、DEV_match を使用して文字列 /adc と構成済みデバイスのリストを照合することにより、デバイスを識別します。そして、文字の残りの部分である 16 が Dxx_open に渡され、ADC が正しいサンプリング周波数にセットされます。

通常、Dxx_open はデバイス固有のオブジェクトを割り当て、このオブジェクトを使用してデバイス・ステートと必要なセマフォが維持されます。終端デバイスの場合は、通常このオブジェクトには 2 つの SEM_Handle セマフォ・ハンドルがあります。その 1 つは、入出力（I/O）オペレーション（SIO_get、SIO_put、SIO_reclaim など）を同期するために使用されます。もう 1 つのハンドルは、デバイスがレディ状態かどうかを判別するために SIO_select で使用されます。デバイス・オブジェクトは、一般に 例 7-27 に示すように定義されます。

例 7-27. Dxx_Obj 構造体

```
typedef struct Dxx_Obj {
    SEM_Handle    sync;     /* synchronize I/O */
    SEM_Handle    ready;    /* used with SIO_select() */
    `other device-specific fields`
} Dxx_obj, *Dxx_Handle;
```

例 7-28 は Dxx_open 用のテンプレートで、終端デバイスの場合の関数の代表的な機能を示しています。

例 7-28.　終端デバイスの代表的な機能

```
Int Dxx_open(DEV_Handle device, String name)
{
    Dxx_Handle objptr;

    /* check mode of device to be opened */
    if ( `device->mode is invalid` ) {
        return (SYS_EMODE);
    }
    /* check device id */
    if ( `device->devid is invalid` ) {
          return (SYS_ENODEV);
    }

    /* if device is already open, return error */
    if ( `device is in use` ) {
        return (SYS_EBUSY);
    }
    /* allocate device-specific object */
    objptr = MEM_alloc(0, sizeof (Dxx_Obj), 0);

    `fill in device-specific fields`
    /*
     * create synchronization semaphore ... */
    objptr->sync = SEM_create( 0 , NULL);
    /* initialize ready semaphore for
SIO_select()/Dxx_ready() */
    objptr->ready = NULL;

    `do any other device-specific initialization required`

    /* assign initialized object */
    device->object = (Ptr)objptr;

    return (SYS_OK);
}
```

最初の 2 つのステップではエラー検査が行われます。たとえば、出力専用デバイスを入力用にオープンするよう要求された場合は、エラー・メッセージが生成されます。また、5 チャネル・システムでチャネル 10 をオープンするよう要求された場合などもエラー・メッセージが生成されます。

次のステップでは、デバイスがすでにオープンされているかどうかが判別されます。多くの場合、すでにオープンされているデバイスを再度オープンすることはできないので、それを要求するとエラー・メッセージが生成されます。

デバイスをオープンできる場合には、Dxx_open の残りの部分は大きく 2 つのオペレーションに分けられます。第 1 のオペレーションでは、デバイス固有オブジェクトが初期化されます。この初期化の一部は、SIO_create から渡された device→params の設定に基づいて行われます。第 2 のオペレーションでは、このオブジェクトが device→object に付加されます。Dxx_open は、SYS_OK を SIO_create に返します。これで、デバイス・オブジェクトが正しく初期化されたことになります。

ハードウェアの動作パラメータをセットするには、構成可能なデバイス・パラメータが使用されます。DSP/BIOS には、どのパラメータを（Dxx_open でなく）Dxx_init でセットする必要があるかについての制約はありません。

オブジェクト・セマフォ objptr→sync は、通常、入出力（I/O）オペレーションの完了時に保留状態になっているタスクを通知するために使用されます。たとえば、タスクが SIO_put を呼び出した結果、objptr→sync で保留状態になりブロックされることがあります。必要な出力が達成されると、objpt→sync で SEM_post が呼び出されます。これにより、Dxx_output でブロックされているタスクが実行可能な状態になります。

DSP/BIOS では、デバイス・ドライバの中で同期セマフォを使用するための特別な制約はありません。このようなセマフォを適切に使用するには、ドライバの要件と基礎ハードウェアの性質によって決まります。

レディ・セマフォ objptr→ready は、Dxx_ready により使用されます。Dxx_ready は、入出力（I/O）用として使用できるデバイスがあるかどうかを判別するために SIO_select により呼び出されます。このセマフォについては、4.7 節「セマフォ」（4-59 ページ）を参照してください。

7.13 リアルタイム入出力（I/O）

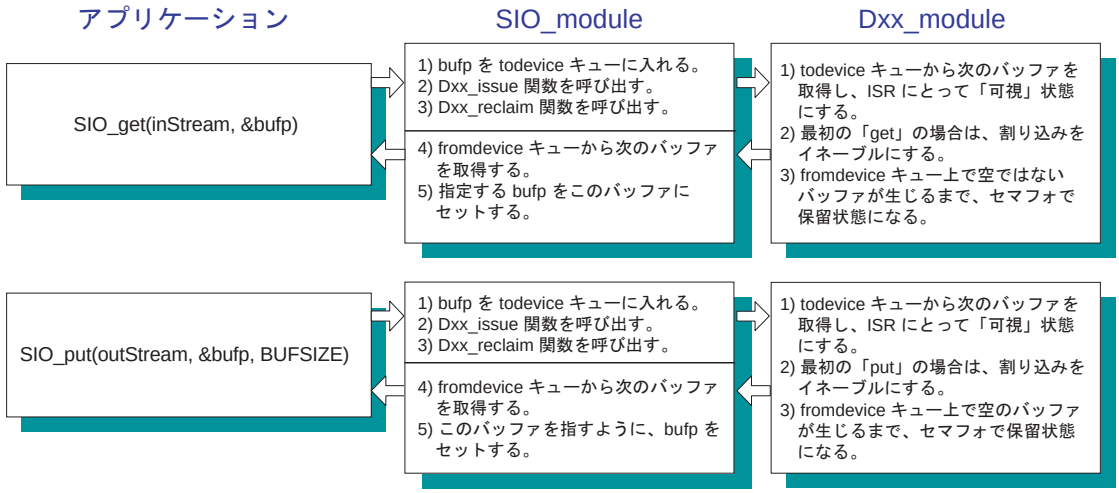
DSP/BIOS には、リアルタイム入出力（I/O）用に使用できるモデルが 2 つあります。DEV_STANDARD ストリーミング・モデルと DEV_ISSUERECLAIM ストリーミング・モデルです。ここでは、これらのモデルについて説明します。

7.13.1 DEV_STANDARD ストリーミング・モデル

DEV_STANDARD ストリーミング・モデルでは、SIO_get を使用して入力ストリームから空でないバッファを取得します。これを取得するために SIO_get は、まず空フレームを device->todevice キューに入れます。次に、SIO_get は Dxx_issue を呼び出します。その結果入出力（I/O）が開始され、保留状態の Dxx_reclaim が呼び出されます。そして、device->fromdevice キューでフル・フレームが使用可能になるまでこのオペレーションが続けられます。このブロッキングを行うために、デバイス・セマフォ objptr->sync に基づいて SEM_pend が呼び出されます。バッファが満杯になるたびに、このセマフォが通知されます。

Dxx_issue は、低レベル・ハードウェア関数を呼び出してデータ入力を開始します。必要量のデータを受信してしまうと、フレームは device->fromdevice に転送されます。一般に、一定量のデータを受信すると、ハードウェア・デバイスは割り込みをトリガします。Dxx は、HWI（図 7-8 の ISR）を媒介としてこの割り込みを取り扱います。HWI はデータを累積して、待機中のフレーム用にさらにデータが必要かどうかを判別します。必要量のデータを受信したと判断した場合、HWI はフレームを device->fromdevice に転送し、デバイス・セマフォに基づいて SEM_post を呼び出します。これにより、Dxx_reclaim でブロックされているタスクが続行可能になります。その後、Dxx_reclaim は SIO_get に戻り、図 7-8 に示すように入力オペレーションが完了します。

図 7-8. DEV_STANDARD ストリーミング・モデルの流れ



objptr->sync は計数セマフォなので、タスクは常にここでブロックされるわけではないことに注意してください。objptr->sync の値は、fromdevice キュー上の使用可能なフレームの数を表します。

7.13.2 DEV_ISSUERECLAIM ストリーミング・モデル

DEV_ISSUERECLAIM ストリーミング・モデルでは、SIO_issue を使用してバッファをストリームに送信します。そのために SIO_issue は、まずフレームを device->todevice キューに入れます。次に Dxx_issue を呼び出して入出力（I/O）を開始した後、戻ります。

Dxx_issue は、低レベル・ハードウェア関数を呼び出して入出力（I/O）を 初期化します。

SIO_reclaim を使用してストリームからバッファを検索します。これを行うために Dxx_reclaim を呼び出します。これは、device->fromdevice キュー上でフレームが使用可能になるまでブロックします。Dxx_issue の場合と同じに、このブロッキングを発生させるために、デバイス・セマフォ objptr->sync で SEM_pend が呼び出されます。デバイス HWI（図 7-9 および 図 7-10 の ISR）が objptr->sync に通知すると、Dxx_reclaim はブロックを解除され、SIO_reclaim に戻ります。次に、SIO_reclaim は device->fromdevice キューからフレームを取得し、バッファを戻します。図 7-9 と 図 7-10 に、このシーケンスを示します。

図 7-9.　ストリームへデータ・バッファを送信する方法

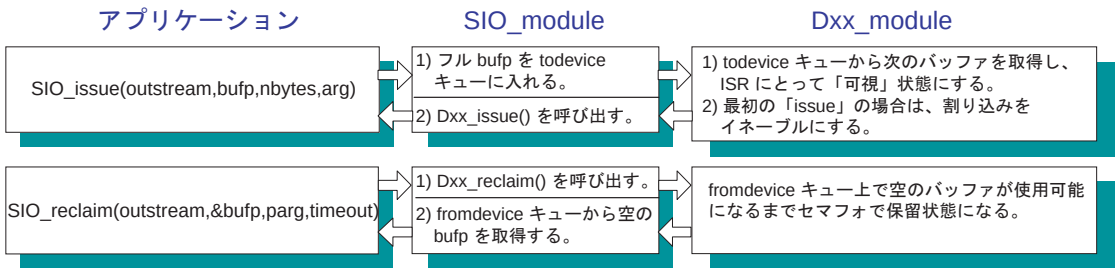
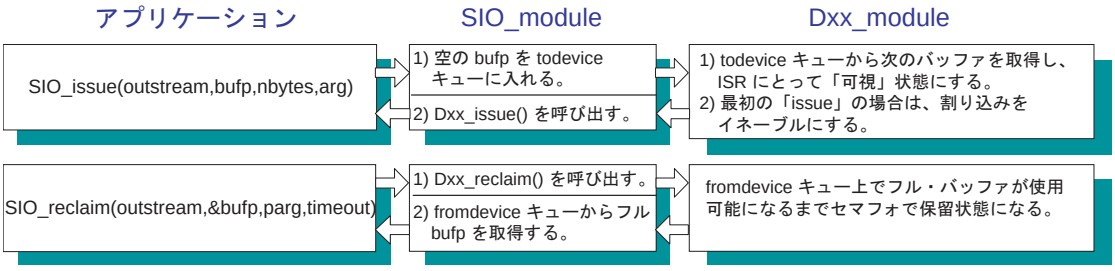


図 7-10.　ストリームからのバッファを検索する方法



例 7-29 は、一般的な終端デバイスの場合の Dxx_issue 用テンプレートです。

例 7-29.　一般的な終端デバイスの場合の Dxx_issue 用テンプレート

```
/*
 * ======== Dxx_issue ========
 */
Int Dxx_issue(DEV_Handle device)
{
    Dxx_Handle objptr = (Dxx_Handle) device->object;

    if ( `device is not operating in correct mode` ) {
        `start the device for correct mode`
    }

    return (SYS_OK);
}
```

Dxx_issue の呼び出しにより、DEV_INPUT または DEV_OUTPUT のどちらかの適切なモードでデバイスが開始されます。デバイスが開始されたことが確認されると、Dxx_issue は単に戻るだけです。実際にデータを操作するのは HWI です。

例 7-30 は、一般的な終端デバイスの場合の Dxx_reclaim 用テンプレートです。

例 7-30.　一般的な終端デバイスの場合の Dxx_reclaim 用テンプレート

```
/*
 * ======== Dxx_reclaim ========
 */
Int Dxx_reclaim(DEV_Handle device)
{
    Dxx_Handle objptr = (Dxx_Handle) device->object;

    if (SEM_pend(objptr->sync, device->timeout)) {
       return (SYS_OK);
    }
    else {   /* SEM_pend() timed out */
       return (SYS_ETIMEOUT);
    }
}
```

Dxx_reclaim が呼び出されると、HWI が device->fromdevice キューにフレームを入れるまで待機状態になり、その後で戻ります。

Dxx_reclaim は SEM_pend を呼び出します。この SEM_pend のタイムアウト値は、静的にまたは SIO_create によりストリームが作成されるときに指定されたものです。バッファが使用可能になる前にタイムアウト時間が過ぎた場合、Dxx_reclaim は SYS_ETIMEOUT を返します。この場合、SIO_reclaim は device->fromdevice キューから何も取得しようとはしません。SIO_reclaim は SYS_ETIMEOUT を返し、バッファは返しません。

7.14 デバイスのクローズ

SIO_delete を呼び出すと、デバイスはクローズされます。その結果、Dxx_idle と Dxx_close が呼び出されます。Dxx_close は、Dxx_idle がデバイスをその初期状態に戻した後でデバイスをクローズします。初期状態は、デバイスがオープンされた直後の状態です。例 7-31 に、この状態を示します。

例 7-31. デバイスをクローズする方法

```
/*
 *  ======== Dxx_idle ========
 */
Int Dxx_idle(DEV_Handle device, Bool flush)
{
    Dxx_Handle objptr = (Dxx_Handle) device->object;
    Uns        post_count;

/*
 * The only time we will wait for all pending data
 * is when the device is in output mode, and flush
 * was not requested.
 */
    if ((device->mode == DEV_OUTPUT) && !flush)         {
/* first, make sure device is started */
        if ( `device is not started` &&
             `device has received data` ) {
             `start the device`
        }

/*
 * wait for all output buffers to be consumed by the
 * output HWI. We need to maintain a count of how many
 * buffers are returned so we can set the semaphore later.
 */
        post_count = 0;
        while (!QUE_empty(device->todevice)) {
           SEM_pend(objptr->sync, SYS_FOREVER);
           post_count++;
        }

      if (`there is a buffer currently in use by the HWI` ) {
           SEM_pend(objptr->sync, SYS_FOREVER);
           post_count++;
        }

        `stop the device`
```

例 7-31.　デバイスをクローズする方法（続き）

```
 /*
  * Don't simply SEM_reset the count here. There is a
  * possibility that the HWI had just completed working on a
  * buffer just before we checked, and we don't want to mess
  * up the semaphore count.
  */
        while (post_count > 0) {
            SEM_post(objptr->sync);
            post_count--;
        }
    }
else {  /* dev->mode = DEV_INPUT or flush was requested */
        `stop the device`


 /*
  * do standard idling, place all frames in fromdevice
  * queue
  */
        while (!QUE_empty(device->todevice)) {
            QUE_put(device->fromdevice,
            QUE_get(device->todevice));
            SEM_post(objptr->sync);
        }
    }

    return (SYS_OK);
}
```

Dxx_idle への引数は、次のとおりです。

```
DEV_Handle  device;    /* driver handle */

Bool        flush;     /* flush indicator */
```

device パラメータは通常、デバイスのこのインスタンスの DEV_Obj を指すポインタです。flush は、デバイスを初期状態に戻すときに、保留データがある場合にどうするかを示すブール・パラメータです。

デバイスが入力モードになっていると、タスクに対してデバイスからデータを検索させる方法がないので、保留データはすべて放棄されます。したがって、入力用にオープンされたデバイスについては flush パラメータに効力はありません。

しかし、出力用にオープンされたデバイスの場合は flush パラメータは重要な意味をもちます。flush が TRUE の場合、保留データはすべて放棄されます。flush が FALSE の場合は、すべての保留データが提供されるまで Dxx_idle 関数は戻りません。

7.15 デバイス制御

SIO_ctrl は、デバイスに対する制御オペレーションを行うために Dxx_ctrl を呼び出します。Dxx_ctrl は一般に、デバイス制御レジスタの内容、あるいは A/D または D/A デバイスのサンプリング・レートを変更するために使用します。Dxx_ctrl は次のように呼び出されます。

```
status = Dxx_ctrl(DEV_Handle device, Uns cmd, Arg arg);
```

- cmd は、デバイス固有コマンドです。
- arg は、オプションのコマンド引数です。

Dxx_ctrl は、制御オペレーションが成功した場合は SYS_OK を返します。そうでない場合は、エラー・コードを返します。

7.16 デバイス・レディ

SIO_select は、デバイスが入出力（I/O）用としてレディ状態にあるかどうかを判別するために Dxx_ready を呼び出します。Dxx_ready は、デバイスがレディ状態であれば TRUE を返し、そうでなければ FALSE を返します。デバイスからバッファを検索するための次の呼び出しがブロックされない状態であれば、デバイスはレディです。通常これは、例 7-32 に示すように、Dxx_ready から戻ったときに device->fromdevice キューに少なくとも 1 つは使用可能なフレームがあることを意味します。SIO_select の詳細は、7.6 節「複数ストリーム間の選択」（7-23 ページ）を参照してください。

例 7-32. デバイスをレディ状態にする方法

```
 Bool Dxx_ready(DEV_Handle dev, SEM_Handle sem)
{
    Dxx_Handle objptr = (Dxx_Handle)device->object;

    /* register the ready semaphore */
    objptr->ready = sem;

    if ((device->mode == DEV_INPUT) &&
       ((device->model == DEV_STANDARD) &&
        `device is not started` )) {
        `start the device`
    }

    /* return TRUE if device is ready */
    return ( `TRUE if device->fromdevice has a frame or
           device won't block` );
}
```

モードが DEV_INPUT である場合は、ストリーミング・モデルは DEV_STANDARD です。デバイスがまだ起動されていない場合は、デバイスが起動されます。これが必要なのは、DEV_STANDARD ストリーミング・モデルでは、アプリケーションは SIO_get の最初の呼び出しの前に SIO_select を呼び出すことができるからです。

デバイスのレディ・セマフォ・ハンドルは、SIO_select から渡されたセマフォ・ハンドルに設定されます。Dxx_ready を明確に理解するために、次に示す SIO_select の詳細事項を検討してみてください。

SIO_select の概要を疑似コードで表すと、例 7-33 のようになります。

例 7-33. SIO_Select 疑似コード

```
/*
 *  ======== SIO_select ========
 */
Uns SIO_select(streamtab, n, timeout)
    SIO_Handle streamtab[];    /* array of streams */
    Int        n;              /* number of streams */
    Uns        timeout;        /*   passed to SEM_pend() */
{
    Int        i;
    Uns        mask = 1;       /* used to build ready mask */
    Uns        ready = 0;      /* bit mask of ready streams */
    SEM_Handle sem;            /* local semaphore */
    SIO_Handle *stream;        /* pointer into streamtab[] */

    /*
     * For efficiency, the "real" SIO_select() doesn't call
     * SEM_create() but instead initializes a SEM_Obj on the
     * current stack.
     */
    sem = SEM_create(0, NULL);

    stream = streamtab;

    for (i = n; i > 0; i--, stream++) {
        /*
         * call each device ready function with 'sem'
         */
        if ( `Dxx_ready(device, sem)` )
            ready = 1;
        }
    }
    if (!ready) {
        /* wait until at least one device is ready */
        SEM_pend(sem, timeout);
    }
    ready = 0;

    stream = streamtab;

    for (i = n; i > 0; i--, stream++) {
        /*
         * Call each device ready function with NULL.
         * When this loop is done, ready will have a bit set
         * for each ready device.
         */
        if ( `Dxx_ready(device, NULL)` )
            ready |= mask;
        }
        mask = mask << 1;
    }

    return (ready);
}
```

SIO_select は、各 Dxx デバイスについて Dxx_ready を 2 回ずつ呼び出します。最初の呼び出しは sem をデバイスに登録するために使用され、2 番目の呼び出し（sem = NULL の場合）は sem を登録解除するために使用されます。

Dxx_ready 関数は、デバイス固有オブジェクト内に sem を保持します（たとえば、objptr->ready = sem）。入出力（I/O）オペレーションが完了したときに（つまりバッファが満杯になるか空になったときに）objptr->ready が NULL でない場合は、SEM_post が呼び出されて objptr->ready をポストします。

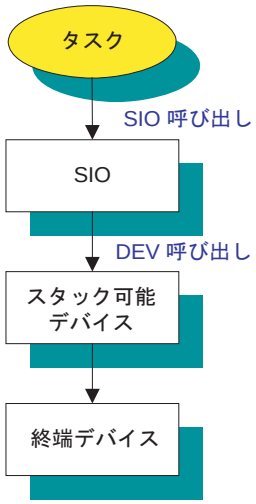
少なくとも 1 つのデバイスがレディ状態にある場合、または timeout が 0 の SIO_select が呼び出された場合、SIO_select はブロックしません。その他の場合は、少なくとも 1 つのデバイスがレディ状態になるかまたはタイムアウト時間が経過するまで、SIO_select はレディ・セマフォ上で保留状態になります。

タイムアウトが発生する前にデバイスがレディ状態になった場合について考えてみましょう。レディ・セマフォは、最初にレディ状態になったデバイスによりポストされます。次に、SIO_select は、各デバイスごとに今度は sem = NULL を指定して再度 Dxx_ready を呼び出します。これには 2 つの効果があります。第 1 に、この後でレディ状態になった Dxx デバイスはレディ・セマフォをポストしません。レディ・セマフォは SIO_select のローカル・メモリ内に保持されているので、もはや存在していないセマフォをデバイスが通知するのを防止することができます。第 2 に、各デバイスを 2 度目にポーリングすることにより、SIO_select は Dxx_ready の最初の呼び出し以降にどのデバイスがレディ状態になったかを判別し、レディ・マスクの中のそれらのデバイスに対応するビットをセットすることができます。

7.17 デバイスの型

デバイスは、大きく分けて 2 種類あります。終端デバイスとスタック可能デバイスです。どちらも同じデバイス関数をエクスポートしますが、実装の方法は少し異なります。終端デバイスは、データ・ソースまたはシンクであるすべてのデバイスです。スタック可能デバイスは、データ・ソースまたはシンクとしての役割をもたず、DEV 関数を使用して他のデバイスとの間でデータの送信または受信を行うすべてのデバイスです。スタッキング・デバイスと終端デバイスがストリーム内にどのように収まるかについては、図 7-11 を参照してください。

図 7-11. デバイスのスタックと終端



スタック可能デバイスを大きく分けると 2 種類あります。インプレース・スタッキング・デバイスと、コピー・スタッキング・デバイスです。インプレース・スタッキング・デバイスは、バッファ内のデータに対するインプレース操作を行います。コピー・スタッキング・デバイスは、データ処理中に、他のバッファにデータを移動します。コピーが必要なのは、受信した量より多くのデータを生成するデバイス（アンパック・デバイスや音声復元ドライバなど）の場合、または出力サンプルを生成するためにバッファ全体にアクセスする必要があり、入力データを上書きできないデバイス（FFT ドライバなど）の場合です。コピー・デバイスは専用のバッファを供給しなければならないことがあるので、これらのタイプのスタッキング・デバイスにはそれぞれ異なる実装が必要です。

図 7-12 に、一般的な終端デバイスの場合のバッファの流れを示します。DSP/BIOS との間の相互作用は、比較的単純です。最も複雑なのは、物理デバイスとの間でやりとりするデータを制御したりストリーミングしたりするためのコードです。

図 7-12.　終端デバイスでのバッファの流れ

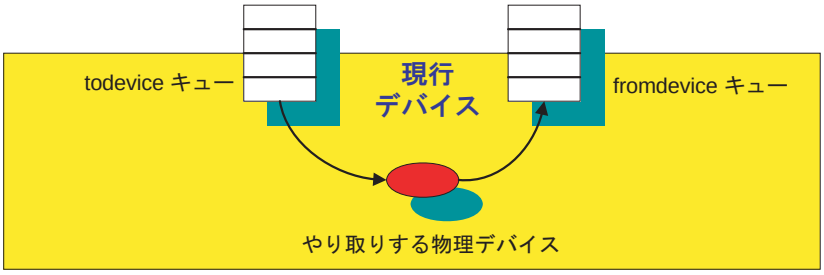


図 7-13 に、インプレース・スタッキング・ドライバのバッファの流れを示します。すべてのデータ処理が 1 つのバッファの中で行われます。これは比較的単純なデバイスですが、コピー・スタッキング・ドライバほど汎用性は高くありません。

図 7-13.　インプレース・スタッキング・ドライバ

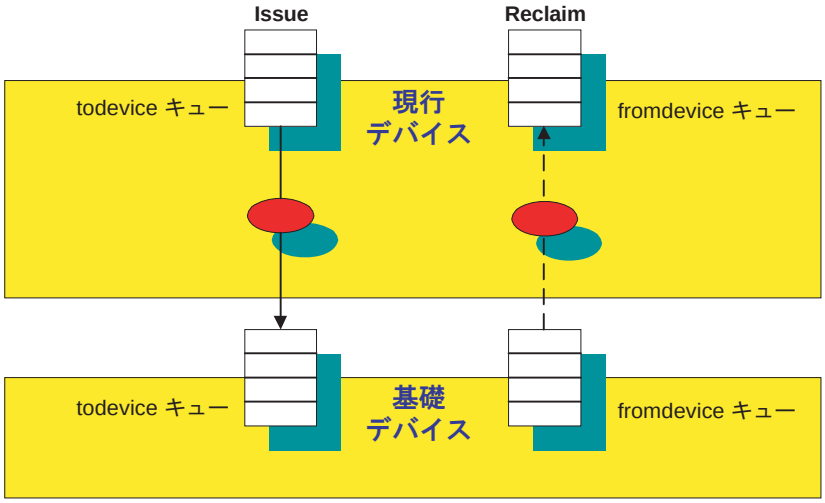
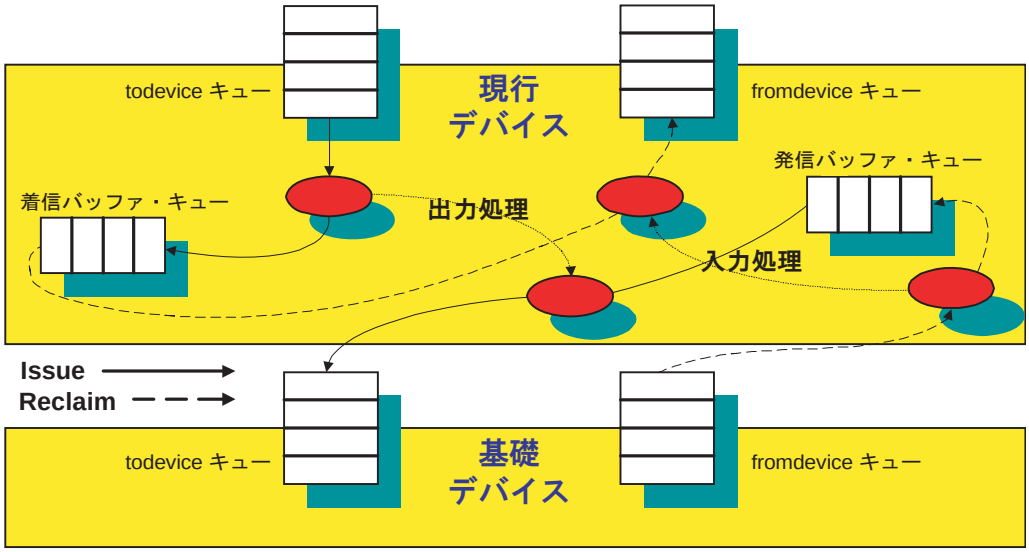


図 7-14 に、コピー・スタッキング・ドライバのバッファの流れを示します。ストリームのタスク側から発生するバッファは、実際にはストリームのデバイス側に移動することはないという点に注意してください。2 つのバッファ・プールは、それぞれ独立した状態を維持します。これは重要なことです。なぜなら、コピー・スタッキング・デバイスでは、タスク側のバッファはデバイス側のバッファとはサイズが異なる場合があるからです。さらに、SIO_ISSUERECLAIM ストリーミング・モデルをサポートできるようにするために、デバイスに入ってくるバッファの順序が不変のまま維持されます。

図 7-14.　コピー・スタッキング・ドライバの流れ



索引

記号

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

あ

い

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

な

に

ね

は

ひ

ふ

へ

ほ

ま

み

め

も

ゆ

よ

ら

り

れ

ろ

わ

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)

■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/

TMS320 DSP/BIOS
ユーザーズ・ガイド

第 2 版 2006 年 9 月
第 1 版 2001 年 12 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
’06.09

日本テキサス・インスツルメンツ株式会社

SPRU570A