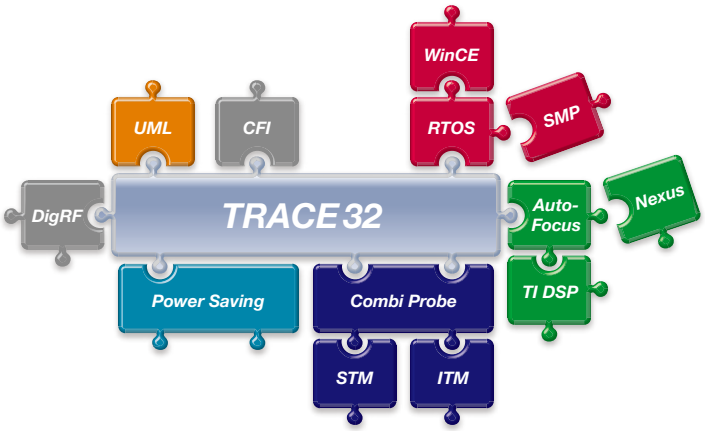
NEWS

News on Microprocessor Development Tools

バックグラウンドで動作するシステム

組み込みシステムは、この 15 年の間に急速に進歩してきました。以前は、一般家庭では使用されないような複雑な製品だけで採用されていましたが、今ではさまざまな製品で採用されています。組み込みシステムはバックグラウンドで動作するため、一般の人は意識することがないかもしれませんが、毎日利用されていることは、組み込みシステム業界では周知の事実です。自動車、携帯電話、MP3 プレーヤー、ホームシアター、洗濯機、調理器具などほとんどすべての製品で、マイクロコントローラ、デジタル信号プロセッサ、およびその対応ソフトウェアを採用した組み込みシステムが日常的に使用されています。そのため、組み込みシステム業界では、システムが完全に問題なく機能するように開発することが求められています。



TRACE32 - 最高の製品

ローターバッハは、組み込みシステムの創成期の 29 年前に創設されました。ローターバッハは、適切なノウハウと多大な努力により、この市場の課題を技術的に解決し、経済的な成功を実現してきました。日常的に利用する機器には、組み込みシステムが不可欠になっています。同様に、組み込みシステムの開発には、TRACE32 が欠かせなくなっています。ヨーロッパ、アメリカ、アジアにおいて、TRACE32 はマイクロプロセッサ開発ツールの世界的ブランドとして、最高の製品と位置づけられています。このことは、2007 年だけでも世界中の 7000 以上の開発現場で TRACE32 が新しく導入されたという事実が証明しています。

継続性と革新性のどちらが重視されるかは、産業分野によって異なります。TRACE32 は、継続性も考慮していますが、主に革新性を重視した製品です。引き続き革新的パートナーとしてローターバッハ製品をご利用いただけるように、2008 年、ローターバッハは積極的な製品開発と安定的成長のための投資を計画しています。

ローターバッハは、2007 年もさまざまなことを達成しました。この 2008 年版ニュースレターでは、当社が昨年達成した技術的進歩を紹介いたします。以下で説明する新機能をお試しいただき、効率的開発にお役立てください。例年お客様をご招待している ESC Silicon Valley で、これらの新機能の多くをご紹介する予定です。

コンテンツ

LAUTERBACH

仮想プロトタイプ用 TRACE32

仮想プロトタイプ

CoWare®	CoWare Virtual Platform	ARM
Synopsys®	Virtual Platform (formerly Virtio)	ARM XScale
VaST™	VaST Virtual System Prototype	ARM OAK PowerPC/eTPU SH2A StarCore TeakLite TriCore/PCP V850

現在では、新製品開発において、短期間で効率的なソフトウェアを開発することが、多くの場合で決定的な要素となっています。開発の初期段階でハードウェアプロトタイプの完成を待つ必要がないように、仮想プロトタイプ（ハードウェアのソフトウェアモデル）の利用が一般的になりつつあります。ターゲットハードウェアの仮想プロトタイプが利用可能になれば、ドライバ、オペレーティングシステム、アプリケーションのデバッグをすぐに開始することができます。

ローターバッハでは、業務用開発環境である TRACE32 をプロジェクトの初期段階で提供できるように、2007 年からソフトウェアモデルのデバッグをサポートしています。ここで使用するデバッグインタフェースは、仮想プロトタイプのデバッグ API です。

GDB フロントエンドとしての TRACE32

GDB フロントエンド

ARM	Available
I386	Available
MIPS32	Available
PowerPC	Available
SH4	Planned
XScale	Available

2007 年初め、TRACE32 GUI を GDB でのプロセスデバッグにも使用できるようになりました。ターゲットハードウェアとの通信インタフェースとして、Ethernet および RS-232 がサポートされています。

アプリケーションプロセスのデバッグ

複数のアプリケーションプロセスを同時にデバッグするために、ローターバッハは独自のデバッグ用エージェント t32server を提供しています。t32server は、Linux のターミナルウィンドウから起動します。これに続いて TRACE32 GUI を使用し、アプリケーションプロセスのテスト用の複数の GDB サーバーを起動できます。t32server サービスは、TRACE32 GUI と GDB サーバーの間のデータ交換を処理します（図 1 を参照）。

その他の新機能

ローターバッハは、2007 年版ニュースレターで、組み込み Linux アプリケーション向け実行 / 停止モード統合型デバッグ機能を発表しました。今年はこれに続けて、SMP Linux のデバッグ用の新しいコンセプトを発表します（13 ページを参照）。

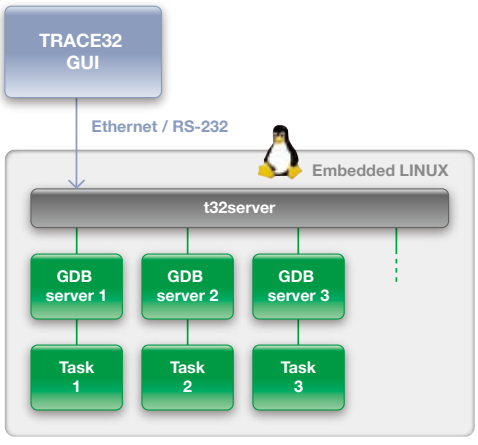


図 1: t32server で 2 つ以上のプロセスを同時にデバッグ

UML で生成したソースと C コードの統合、テスト、デバッグ

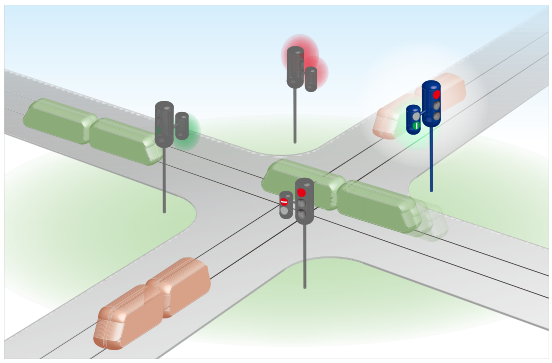


図 2: 交差点に電車用信号を追加

概要

ソフトウェアの複雑さは増すばかりで、将来的には CASE ツールなしで完全な管理を行うことは不可能になるでしょう。とりわけ UML の登場で、モジュラーソフトウェアコンポーネントの統一設計が可能になり、自動コード生成によって開発当初から一般的なコーディングエラーを防止できるようになりました。

しかし、こうした新技術への移行には手間がかかると、多くの企業が移行を躊躇しています。企業には長年にわたって構築してきたコードベースがあります。一方で、ソフトウェアは拡張や改善を繰り返しています。これは、いわゆるスパゲッティコードの増加の原因にもなっています。移行するには、機能している数百万行のコードを解析して再プログラム化し、最後に再テストする必要があるとされています。

ほとんどの場合、こうした懸念には根拠がありません。ほぼ変更の必要はなく、既存の C コードを UML モデル要素に移行することができます。新しい拡張は UML モデルで記述して生成し、既存の C コードと統合できます。そのため、C から UML への段階的な移行が簡単に実現できます。実際の手順は以下のようになります。

1. 既存の C コードから UML へ移行
2. UML で記述した新機能を統合
3. UML、C++、C で統一的にテストとデバッグを実行

既存の C コード

まず、C コードで記述したプロジェクト例を説明します。

次に示す例は、交通信号システムを記述したものです（交通信号システムは単純であるため、ソフトウェア設計者が例として好んで採用しています）。4 基の信号機がある一般的な交差点を考えてみましょう。向かい合う 2 基の信号機は、同時に制御されます。これらの信号機に対して直角に位置する信号機は、動作が相補的になるように制御されます。信号機は、赤、赤 / 黄、緑、黄、赤と変わります。ある方向の信号機 2 基が赤の間は、通行権は反対方向に移ります。その結果、全ての方向の信号機が短時間だけ赤になります（これは重要な点なので、後で説明します）。

この回路は、C では連続ループ（while (1)）内の単純な制御構造（switch-case）として記述されます。この連続ループはアプリケーションの main() ルーチンに直接記述されており、初期化直後に呼び出されます（図 3 を参照）。

新しい要件

交通計画で、両方向のトラム（3 ページの図 2（縦方向と横方向））が交差点を横断することが決定したと

```
...
int main (void)
{
   horizontal = vertical = red;
   state = closed_to_h;

   while (1)
   {
      switch (state)
      {
          case closed_to_h:
             wait (1);
             horizontal = yellowred;
             state = yellowred_h;
             break;

          case yellowred_h:
             wait (3);
             horizontal = green
             state = green_h;
             break;
          ...
```

図 3: 既存の交通信号制御用 C コード；“main()” 関数には CASE 文で組まれた再帰プログラム

します。そのために、トラムを専用の信号機を使って交差点前で停止させる必要があります。 停止後、トラム運転手からの要求に従い、次に赤信号になったときに自動車の交差点への進入を禁止し、トラム側に通

行権を許可します。

C コード用 UML ラッパー

もちろん、これらの要件を C コードにパッチとして適用することもできます。ただしその場合、関数が大きく複雑になります。代わりに、モジュラーアプローチを採用し、UML を利用して問題を解決します。

まず、既存のアプリケーションを UML モデルに移行します。この作業はそれほど難しくありません。既存のアプリケーション全体を含む「CarJunction」というクラスを作成します（図 4 を参照）。

調整が必要な箇所は、対応するインタフェースだけです。main() 内の通常の連続ループは、すでにフレームワークの一部となっており、省略する必要があります。または、「main()」を「processJunction()」という名称に変更して、新しいコードの開始点とします。この関数は、すべての信号機が赤になった時点で終了します。ここで、必要に応じて赤信号の間の追加処理を実行できます（図 5 を参照）。これで移行作業は終了です。

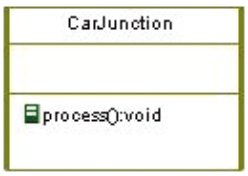


図 4: 全ての旧アプリケーションが含まれている "CarJunction" クラス

```
void processJunction (void)
{
   horizontal = vertical = red;
   state = closed_to_h;

   do
   {
      switch (state)
      {
          case closed_to_h:
             wait (1);
             horizontal = yellowred
             state = yellowred_h;
             break;
  /*........*/
          case yellow_v:
             wait (3);
             vertical = red;
             state = closed_to_h;
             break;

      } // switch (state)
   } while ((state != closed_to_h)
         && (state != closed_to_v));
} // processJunction()
```

図 5:UML ツールに統合されるように修正された C コード

新要件のための UML の設計

では、新要件のための設計を行います。「Junction」というクラスを作成します。このクラスには、以前の信号機制御に対応する CarJunction と、新しいトラム用信号機 2 基が含まれます（図 6）。設計はこれで終了です。次に、状態遷移図を使用して拡張した交差点での挙動を重点的にみていきます。

ここでも、既存のアプリケーションを開始点として、その挙動全体を新しい交差点の 1 つの状態（状態 :cars）として統合します（次ページの図 7 を参照）。新モデルがこの状態になった場合は、既存のアプリケーションと完全に同一の挙動を示します。新要件（トラム運転手による通行権の要求）が必要な場合にのみ、既存のコードが終了して新モデルが開始します。両方向のトラム線について、各段階とその遷移：- 要求、- 待機、- 開放、- 待機（通行を許可するトラム線の切り替えなど）を、状態遷移図にまとめます。これで新要件のモデル作成は完了です。

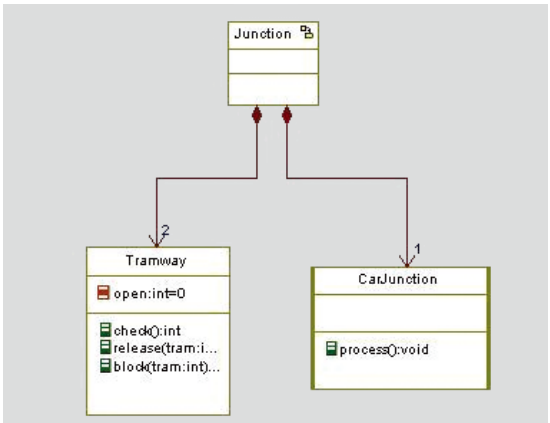


図 6: 旧信号スイッチプログラムと新たに設置された交通信号が統合された "Junction" クラス

コード生成

ここから、実行可能なアプリケーションを生成することができます。優れた UML ツールであれば、それだけで上記のモデルの完成アプリケーションを生成できます。ただし、統合についていくつか注意点があります。この例では、最初のアプリケーションは C で記述されており、新モデルでもそのまま使用されています。しかし、UML ツールは通常 C++ コードを生成するため、

LAUTERBACH

新しい CarJunction は、C コードを参照するラッパークラスになります。ここで、C と C++ の間での通常の調整を考慮する必要があります。

一般に、RTOS はモデルの管理に使用され、必要に応じて統合します。統合後は、ボタンをクリックするだけで、実行可能アプリケーションがモデルから生成されます。

ターゲット実行

組み込み技術分野では、多くの場合、ターゲットハードウェアと開発に使用するコンピュータが異なります。そのため、生成したアプリケーションをいわゆる「ターゲット」にロードして実行する必要があります。これは、ターゲット上にコードをインストールする外部ツールを使用して実現できます。

これは、Telelogic 社の UML ツール Rhapsody とローターバッハの TRACE32 デバッガのように、UML ツールとデバッガが共通（ソフトウェア）インタフェースで通信する場合は特に簡単です。この場合、モデリングツールでボタンをクリックすると、デバッグインタフェース経由でターゲットにアプリケーションがロードされ、必要であればそのまま実行されます。設計、モデリング、生成、実行のサイクル全体を、1 つの GUI で操作することができます。

C でのテスト

初めに説明したように、テストおよびデバッグを異種混在環境で行うには一定の要件があります。アプリケーション全体を関数および実装ごとにテストできるようにする必要があります。

ここでは、C で記述した動作しているアプリケーションを想定していますが、そのコードもデバッグできる必要があります。つまり、既存のコードに新機能を移行してテストし、古いソースコードを維持する必要があります。少なくともアプリケーションのこの部分については、組み込み技術分野では C コードのデバッグが一般的になっているため、任意の市販デバッガを利用できます。

C++ でのテスト

ソースレベルでの新機能のテストプロセスは、ますます興味深いものになっています。ほとんどの UML ツールは、テンプレート、ポリモーフィズム、例外処理といった特徴をすべて含む C++ コードを生成します。ここで、使用するデバッガでコンパイラの C++ 方言を完全サポートする必要があることに注意してください。ローターバッハの TRACE32 デバッガはすべての最新 C++ コンパイラをサポートしているため、オブジェクト指向 C++ コードを効率的にデバッグできます。

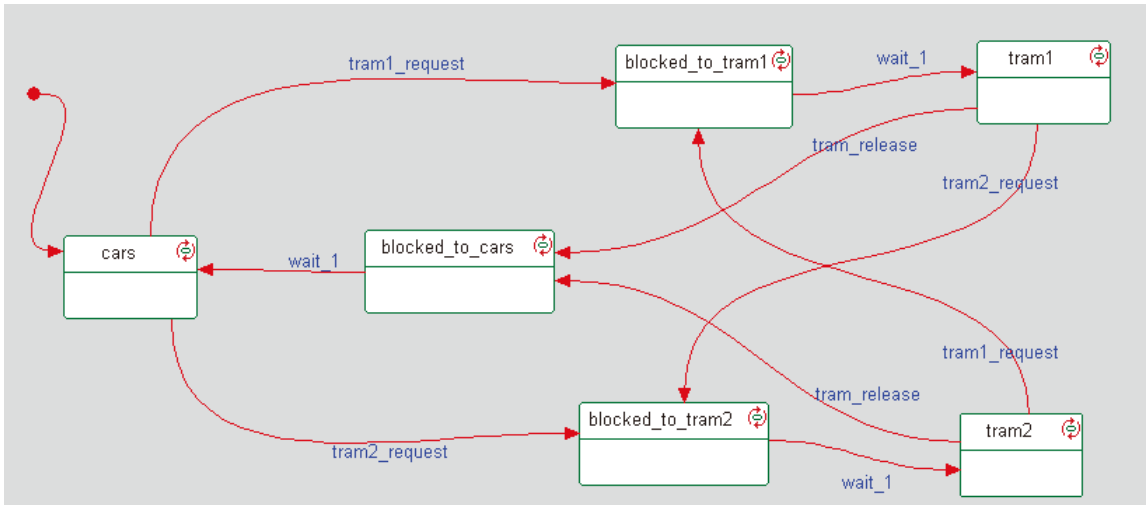


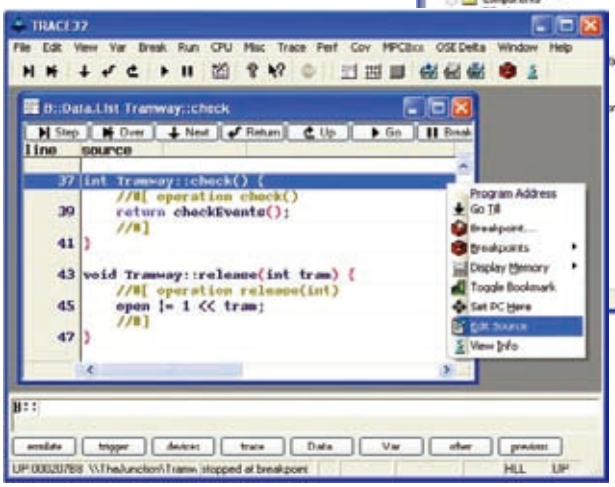
図 7: 新しい交通信号スイッチのステートダイアグラム

UML でのテスト

しかし、ここでは C++ ではなく UML でコードを記述しています。この場合、モデリングレベルでデバッグを実行するよりも効果的な方法は何でしょうか。

Telelogic 社の Rhapsody は、この用途に適したオプションを複数装備しています。

アプリケーションの挙動を UML で完全にモデリングする場合、モデルからシーケンスを直接シミュレートできます。単純な挙動分析の場合、実際のターゲットハードウェアは必要ありません。また、ターゲットではシミュレーションとタイミングが異なるため、アプリケーションを「アニメーション」として実際のターゲットで実行することもできます。UML ツールは、通信チャネル（シリアルまたは Ethernet）経由でターゲットプロセスを制御 / 視覚化します。これにより、状態遷移図の各段階をステップ実行する、あるいはイベントを発生させることができます。



TRACE32 デバッガとの組み合わせでは、他に利用可能なインタフェースがない場合、この通信をデバッグインタフェースで置き換えることができます。「アニメーション」により、UML モデルでの直接デバッグが可能です。

UML、C++、C の統合型デバッグ

この例では、コードが UML、C++、C の 3 つのレベルに分かれています。特に C++ コードでエラーが見つかった場合は、生成された C++ コード自体ではなくモデル内で修正する方がお勧めです。一方、モデル要素の実装をデバッグする場合は、検索が不要になるため、ソースで行うのが最適です。

TRACE32 と Rhapsody を統合することで、簡単な相互制御など、これらのレベルをまとめて管理するためのさまざまな機能が提供されます（図 8）。たとえば、Rhapsody でモデル要素を 1 回クリックするだけで、TRACE32 で対応ソースコードが表示されます。これにより、変数、関数呼び出しなどを詳細に分析できます。また、デバッガでソースコード内の行をクリックすると、対応するモデル要素が UML ツールで強調表示されます。この機能は、デバッガで問題が見つかり、その問題をモデルで修正する必要がある場合に特に便利です。面倒な検索の後に該当要素を手動で表示させる必要がありません。

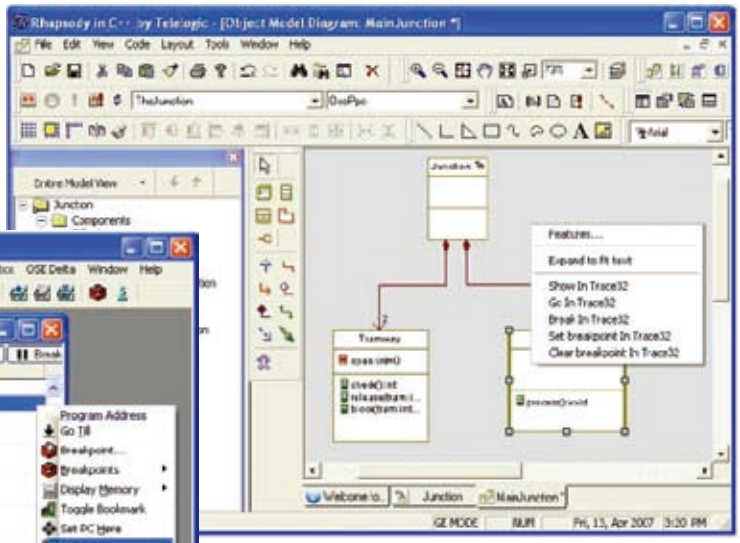


図 8:TRACE32 と Rhapsody の統合化により、UML、C++ および C によるデバッグが実現

また、デバッガのブレークポイントをモデル内で定義し、ターゲットを Go および Stop 状態に設定することもできます。これにより、クラスでブレークポイントを設定してターゲットを実行するという操作をすべて UML ツールから実行できます。検査対象のモデル要素が呼び出されると、デバッガがアプリケーションを停止します。

まとめ

完全な再設計が常に必要というわけではありません。特に、時間や組織的な制約からリエンジニアリングが不可能な場合、段階的なアプローチが適していることが多々あります。このアプローチでは、必要な作業が少なく、最新の革新的な CASE ツールに既存のソフトウェアを段階的にインポートできます。

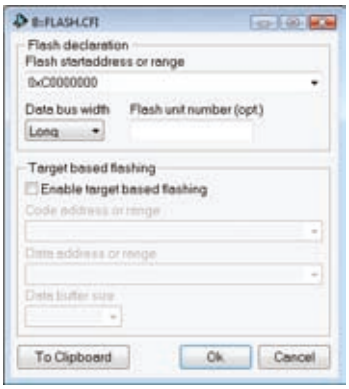
このプロセスで使用するツールが実際に役立ち、作業の障害にならないことは当然のこととして求められますが、それには適切なツールの選択が重要です。これらのタスクを単に実行できるだけではなく、その効率を向上できることが要求されます。

ローターバッハと Telelogic 社は、TRACE32 と Rhapsody という互いのツールを統合することでこれを実現しました。また、複雑さが増し、要求事項も増えているソフトウェアの品質を保持できるように、開発者の支援を行っています。今では、既存のコードベースを UML で簡単に新しくすることができます。

最新ニュース（概要）

CFI フラッシュプログラミング

2007 年 9 月、共通フラッシュメモリインタフェース（CFI）を使用したフラッシュ宣言の自動生成がサポートされました。



これまでは、フラッシュをプログラムするには、デバイスの正確なタイプと構成についての詳細な知識が必要でした。CFI 準拠のフラッシュデバイスの特別な問い合わせモードにより、TRACE32 でプログラミングに必要なパラメータを特定し、このデータを使用してフラッシュ宣言を自動的に作成できるようになりました。

シリアルフラッシュ /NAND フラッシュ

2007 年 9 月以降、TRACE32 では、専用のコマンド（FLASHFILE）による NAND フラッシュデバイスとシリアルフラッシュデバイスのプログラミングがサポートされています。

コードオーバーレイ

2007 年 12 月以降、TRACE32 では、sYmbol.OVERLAY コマンドを使用した ELF ファイル内のコードオーバーレイ用シンボル管理がサポートされています。現在、ARC、ARM、PowerPC の各アーキテクチャがサポートされています。2008 年には対応アーキテクチャがさらに追加される予定です。

PCP デバッガ

2007 年 11 月以降、Infineon 社の TriCore プロセッサの周辺制御プロセッサ（PCP）をデバッグするには、別の TRACE32 インスタンスの起動が必要になりました。これは、PCP 用の C コードのロードとテストを有効にするために必要な変更です。このインスタンスは、PCP トレースの設定およびトレース情報の分析にも使用します。PCP のデバッグおよびトレース用ライセンスは、TriCore 用 TRACE32 デバッガに含まれています。

新たなサポートプロセッサ

New architectures

P.A. Semi	PWRficient™	Q2/2008
Renesas	H8SX	Available
Tensilica	Xtensa Processors	Available
Texas Instruments	TMS320™C28xx	Available

PWRficient

PA6T-1682M デュアルコアプロセッサは、ローターバッハのデバッガで初めてサポートされたPWRficientアーキテクチャの 64 ビット PowerPC です。

TMS320C28xx

TMS320C28xx DSP 用デバッガにより、Texas Instruments 社の TMS320 プロセッサプラットフォームに対するローターバッハのサポートがさらに広がりました。

New derivatives

AMCC	PPC405 - PPC405EX/EXR PPC44x - PPC460EX
ARC	ARC - ARC® 700 Core - ARCtangent-A4 - ARCtangent-A5
Freescale™	ColdFire - MCF52110/MCF52100 - MCF5221x - MCF523x/MCF532x - MCF5372/MCF5373 MPC5500 - MPC551x - MPC56xx MPC5200 - MPC512x with AXE audio processor PowerQUICC II - MPC8377 PowerQUICC III - MPC8572

Freescale™ (cont.)	ARM11/StarCore - MXC91321
Infineon	C166S - XC22xx/XC23xx/XC27xx - XE166 TriCore - TC1736 - TC1767/TC1767ED - TC1797/TC1797ED
Microchip	MIPS32 - PIC32™
NXP	ARM9 - LPC3000 MIPS32 - PNX83xx/PNX85xx
STMicro-electronics	MPC5500 - SPC563Mxx - SPC560Bxx/Pxx/Sxx Cortex-M - STM32
Texas Instruments	Cortex-A/DSP - OMAP3430

LAUTERBACH

ARMコアの節電モードでのデバッグ

エネルギーの効率的利用および待機時間の延長に対する要望の増加に応え、多くの場合、最新のコアではさまざまな節電メカニズムが採用されています。これらのメカニズムをソフトウェアで利用し、必要に応じてコアの周波数を下げる、またはコアを完全に停止させることができます。ただし、いわゆる節電モードを使用する場合、デバッガとコアの間の通信に問題が生じることがあります。以下で、ARMアーキテクチャを例に、節電モードを使用するプログラムを問題なくデバッグする方法をいくつか紹介します。

節電モードの無効化

デバッグ中は節電モードを無効にするのが最も簡単な方法です。たとえば、ARM11 にはいわゆる Powerdown Disable Bit があります。このビットをセットすると、外部電源およびクロック管理にデバッグ実行が通知されます。プログラムが節電モードを有効化しようとしても、電源およびクロック管理がそれを無視し、デバッグ中に問題が発生するのを回避します。ARM11、Cortex-A、Cortex-M に対応した TRACE32 デバッガでは、Powerdown Disable Bit のセットが基本的な対応の 1 つになります。

もちろん、節電モード関連の設計の挙動をテストする場合は、デバッガオプション（SYStem.Option PWRDWN ON、図 9 を参照）で変更できます。

図 9: デバッグ中の省電力モードを無視するか否かを "PWRDWN" オプションで指定

ただし、Powerdown Disable Bit がない ARM ベースのコアも多数あります。この場合は、節電モードを有効にする命令を記述していないデバッグ版ソフトウェアを作成します。しかし、この方法には、デバッグ版ソフトウェアとリリース版が異なるという問題があります。

節電モードのデバッグおよび使用が問題につながる理由は何でしょうか。このパートでは、まずシングルコアチップの場合について説明します。（コアの）電源を切る、あるいはクロックを停止することが、デバッグにどのように影響するかを説明します。次に、マルチコアチップの場合について説明します。発生する可能性のある問題の中から、1 つのコアのクロックを停止することによるチップ全体のデバッグへの影響について特に説明します。

電源切断

デバッガでは、JTAG コネクタの VTREF ピン（ターゲット電圧基準ピン）で電源切断を検出できます。デバッガでこの節電方法を使用するように設定した場合、デバッガはエラーメッセージを出力する代わりに、電源投入まで待機します（StandBy モード）。

実際には、デバッガは以下の 2 つのいずれかの状態になります：

- Up（StandBy）: プロセッサに電力が供給され、デバッガとプロセッサの間で通信が行われています。
- StandBy: 電源が切断され、デバッガは SRST ライン経由で ARM プロセッサをリセット状態に保ち、電源が再投入されるまで待機します（図 10 を参照）。

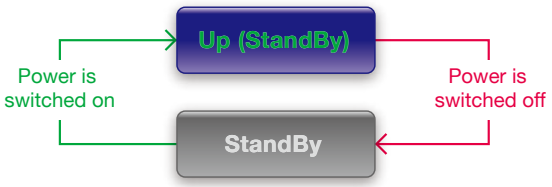


図 10: パワーダウン後、エラーメッセージを発行する代わりに、電力供給再開まで待機

電源が再投入されると、デバッガはリセットラインを解放し、プロセッサは電源投入によるリセット後と同様に実行されます。

ただし、ARM プロセッサの電源投入によるリセットで、デバッグロジック、ETM/ETB ロジック、ETM ピンの割り当てがリセットされます。電源切断前に有効になっていた設定は失われます。

ARM プロセッサがプログラムの実行を開始する前にデバッガがこれらの設定を復元できた場合に限り、デバッグをシームレスに継続できます。

電源投入時に必要な復元処理をデバッガで実行するには、ターゲットハードウェアがいくつかの要件を満たしている必要があります。

ベストケース

ベストケースは、以下の要件が満たされている場合で

す。

1. デバッガが SRST ライン経由で ARM プロセッサをリセット状態に保持できる。
2. プロセッサリセット（SRST）およびデバッグロジックのリセット（TRST）が別ラインである。
3. JTAG インタフェースの VTREF ピン経由で、デバッガが電源切断をすぐに検出できる。

これらの要件を満たしている場合、デバッガは電源投入の検出時にデバッグロジックのリセットだけを解放します。デバッグロジックの設定および ETM/ETB 設定は復元されます。次に、プロセッサリセットが解放され、多少の遅れはありますが、デバッグをシームレスに継続できます（図 11 を参照）。

I/O ピンはメモリマッピングされたレジスタ経由で設定されるため、プロセッサリセットが解放されてプログラムが開始する前に ETM トレースポート用に復元することはできません。したがって、プログラムによって ETM ポートをできるだけ早く有効にする必要があります。

その他のケース

要件 1 と 2 のいずれかまたは両方が満たされていない場合、ARM プロセッサが電源投入時に起動し（図 12 を参照）、デバッガがデバッグロジックと ETM/ETB 設定を同時に復元します。これは、望ましくない結果をまねき、同時にオンチップのブレークポイントおよびトレースブロードキャストが復元フェーズで無効になっています。

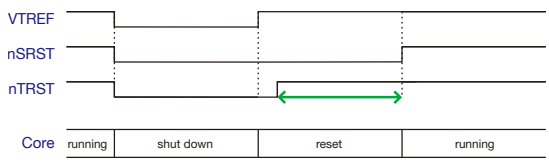
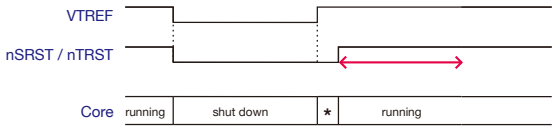


図 11: 次の電力再開時，まずデバッグロジックと ETM/ETB 設定がリストア（緑の矢印）. その後、ARM プロセッサはプログラム実行を再開 .

クロック供給を停止

デバッガでは、プロセッサクロック停止は全く別の問題です。ここでは、デバッガとプロセッサの間の通信、具体的にはデバッガとオンチップデバッグロジックの間の通信について説明します。この通信は、ARM プロセッサがプログラムを処理している間も含め、常に行われています。



(* = reset)

図 12: デバッグロジックおよび ETM/ETB 設定の復元中（赤い矢印）にプロセッサ動作の再開は思わぬ誤動作の原因に .

デバッグロジックがプロセッサクロックから独立している場合、プロセッサクロックを停止しても通信は機能します。ただし、（非同期に標準化された）デバッグロジックのクロッキングでは、しばしば遅延 / 分周されたプロセッサクロックが使用されます。デバッガの設定で SYStem.JtagClock RTCK コマンドを使用する必要がある場合がこれに該当します。

以下の 2 つの条件を満たす場合、プロセッサクロックを停止すると、デバッガとオンチップデバッグロジックの間の通信に影響することがあります :

1. デバッグロジッククロックがプロセッサクロックに依存する。
2. デバッガとオンチップデバッグロジックの通信中にプロセッサクロックを停止する。

ここで、プロセッサクロックを停止すると、以下のようになります :

1. ARM プロセッサが現在の状態をフリーズします。
2. デバッグロジックのクロックはプロセッサクロックに依存するため、デバッグロジックも現在の状態をフリーズします。
3. オンチップデバッグロジックが応答しなくなるため、デバッガがタイムアウト後にオンチップデバッグロジックとの通信を中断します。

プロセッサクロックが再度有効になると、デバッグロジックがフリーズ時点からデバッガとの通信を再開します。一方で、デバッガはデバッグロジックが応答しなくなったときから、通信を再初期化しています。

ARM プロセッサのデバッグロジックはデバッガにより制御されるため、デバッガが通信を再度同期させるには、デバッグロジックをリセットする必要があります。しかし、プログラム実行中の少なくとも短時間オンチップブレークポイントがすべて削除されます。これは最適な状態ではありません。

信頼性の高い解決方法として、以下のものがあります：

1. プロセッサクロックをチェックする

オンチップデバッグロジックとの通信の開始前に、プロセッサクロックが有効かどうかをチェックするようにデバッガを設定することをお勧めします。その場合、デバッガは各通信の終了ごとにデバッグロジックを Run Test/Idle モードに設定します。このモードでは、通信の再開前にプロセッサクロックを簡単に検出することができます。

2. 状態クエリーを削減する

ARM コアがプログラムを処理している間、デバッガはデバッグロジックに対して状態クエリーを 1 秒間に 10 回送信します。これにより、プログラムがまだ実行されているか、あるいは停止したかをチェックします。この解決方法では、この状態クエリーの頻度を減らします。クエリー頻度としては、2 ～ 3 秒間に 1 回を推奨します（SETUP.URATE 3.s）。この方法は単純で、クエリーが少なくすることで、通信問題の発生のリスクを低下させます。

3. タイムアウトを長くする

プログラムが非常に頻繁に（ただし短時間）プロセッサクロックを停止する場合は、デバッグロジックとの通信を中断するまでの待機時間をより長くデバッガに設定します（SYStem.POLLING SLOW）。状況によっては、プロセッサクロックの停止がデバッガで検出されないこともあります。

マルチコアチップ

マルチコアチップは、デイジーチェーン接続されているコアのデバッグケーブルで 1 つだけ JTAG インタフェースを使用しますが、個々のコアで節電モードを使用する場合は追加の手段が必要になります。たとえば、デバッグロジックのクロッキングがコア 2 のコアクロック（RTCK）に依存しているときに、このコアのみクロックが停止したとします（図 13）。

「クロックの停止」で説明したように、この場合はコア 2 のデバッグロジックがフリーズします。このとき、コアのデイジーチェーン接続により、コア 1/ コア 3 のデバッグコマンドがコア 2 を経由する必要があるため、チップ全体のデバッグがブロックされます。

コア 2 のクロックが再度有効になったときに問題なくデバッグを継続できるように、デバッグロジックとの通信の開始前に、個々のコアのデバッガを、コアクロックが有効かどうかをチェックするように設定することをお勧めします。

ただし、1 つのコアのデバッグロジックがフリーズするとすぐにチップ全体のデバッグがブロックされるという問題に対しては、この方法では解決になりません。現在この問題に対する解決方法はありますが、チップメーカーは自社の解決方法を機密扱いとしているため、ここで紹介することはできません。

そのため、以下では、CoreSight DAP 技術によるこの

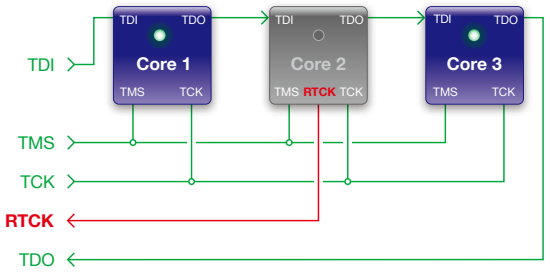


図 13: チップ全体のデバッグ時 Core2 ブロックのクロックを停止

問題の解決方法のみを紹介します。

CoreSight 技術によるデバッグおよびトレース機能を採用したマルチコアチップの場合、TRACE32 デバッガは個々のコアのデバッグロジックと直接通信せず、代わりにいわゆる Debug Access Port（DAP）と通信します。DAP は、デバッグコマンドを個々のコアに割り振る処理を行います。

節電モードのデバッグについて DAP 技術の最も重大なイノベーションは、DAP 専用の電源およびクロックドメインがあることです。つまり、DAP は個々のコアの節電モードの影響を受けないということです。同時に、CoreSight では、コアがデイジーチェーン接続されることはありません。その代わり、以下のように制御されます：

- デバッグロジックがメモリマッピングされたデバッグレジスタを使って実装されている場合、デバッグロジックはバスアクセスで制御されます。
- コアが従来のデバッグロジックで動作している場合、コアのデバッグロジックは JTAG アクセスポート経由で制御されます。

このような新しい制御メカニズムにより、DAP は常に個々のコアを直接指定でき、フリーズしたコアによりデバッグコマンドがブロックされることはありません（図 14 を参照）。

一般に、開発段階でマルチコアチップでの節電モードのデバッグが特に重要な場合は、チップの選択時に半導体メーカーとこの問題について検討する必要があります。詳しくは、ローターバッハの技術者にお問い合わせください。

エネルギー消費の分析

ローターバッハは、ETM およびオフチップのトレースポートを持つ ARM ベースのチップについて、節電モードが設計に対してエネルギー消費の削減に実際に貢献しているかどうかを分析するオプションを提供しています。この分析には、選択した測定点の電流と電圧を記録し、その結果とトレースポートで出力されたプログラムフローの情報の関係を調べる測定方式が必要です。

ローターバッハは 2007 年より、そのような測定方式を提供しています。測定には、TRACE32 デバッガに加えて、以下が必要です :

- ETM 用リアルタイムトレース
- アナログプローブ
- ローターバッハ製ロジックアナライザ (PowerTrace II の IProbe Logic Analyzer または TRACE32-PowerIntegrator)

この測定方式では、電流 / 電圧とプログラムフローの両方に同期タイムスタンプが TRACE32 ソフトウェアで記録されるため、制御ソフトウェアのフローと電流 / 電力消費の関係を簡単に表示して分析できます。図 15 に、測定期間内に実行された各関数について、合計エネルギー消費の絶対値と合計エネルギー消費に占める割合を表示した例を示します。この方法で、重要なエネルギー消費と待機時間データを検証できます。

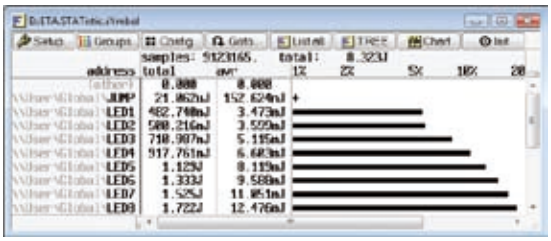


図 15: プログラムとハードウェアの電流 / 電力消費の関係をグラフィカルに表示

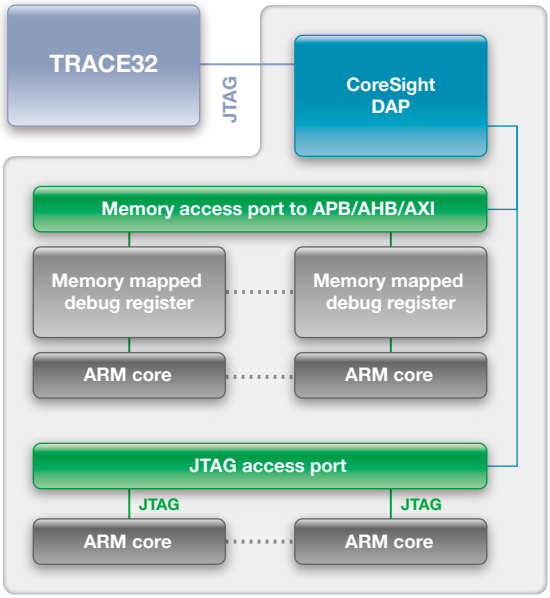


図 14:DAP は常に直接個々のコアのデバッグロジックにアクセス可能 — 個別のコアのクロックにコアが供給されている否かにかかわらず、チップ全体のデバッグが常に可能

LAUTERBACH

対称型マルチプロセッシング（SMP）対応の新しいデバッグコンセプト

高品質デバッガおよびリアルタイムトレースツールの世界的なトップメーカーであるローターバッハは、ESC 2008 で SMP システム対応の新しいデバッグコンセプトを発表します。SMP システムは、オペレーティングシステムを使用して、プロセスを複数のコアまたはハードウェアスレッドに動的に分散させます。ESC 2008 では、MIPS 34K で動作する SMP Linux のデモンストレーションを行う予定です。

ハードウェアの並列化

複雑なアプリケーションでプロセッサ性能を向上させ、エネルギー消費を抑えるため、タスクの並列化が多く使用されています。最も一般的なのが、同一タスクを実行できる複数の同一実行ユニットを使用する手法です。

ほとんどのユーザーにとって、「同一実行ユニット」とは対称型マルチコアプロセッサを意味します。その代表的な例として、1 つのコアで恒常的に実行されているオペレーティングシステムのカーネルが、アプリケーションプロセスをすべてのコアに均等に分散させる場合があります（図 16）。

ただし、タスクの並列化にマルチコアプロセッサは必須ではありません。たとえば、シングルコアプロセッサでも並列化を実現する手法として、ハードウェアによるマルチスレッド化があります。ここでは、パイプラインアーキテクチャのコアでの基本的な問題について説明します：キャッシュミスや命令間のデータ依存がある場合、必要なデータが利用可能になるまで命令のパイプライン処理をストールする必要があります。命令処理時間とメモリアクセス時間の差が大きいほど、性能も低下します。

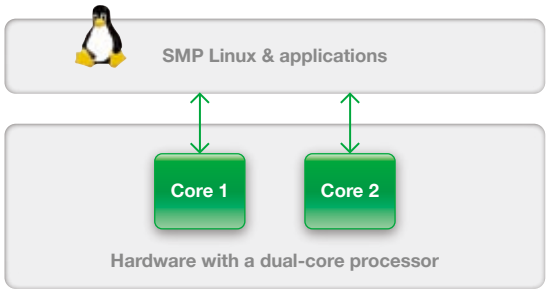


図 16: デュアルコアプロセッサで SMP Linux

ハードウェアによるマルチスレッド化では、コアプロセスを複数の独立タスクとして擬似的に同時処理することで、このような状況に対応します。オペレーティングシステムの場合、ここでいうタスクとプロセスは同義です。

基本的な処理は、以下のとおりです。必要なデータがまだ利用可能でないためにタスクの処理を進めることができなくなると、別のタスクの処理が続けて実行されます。

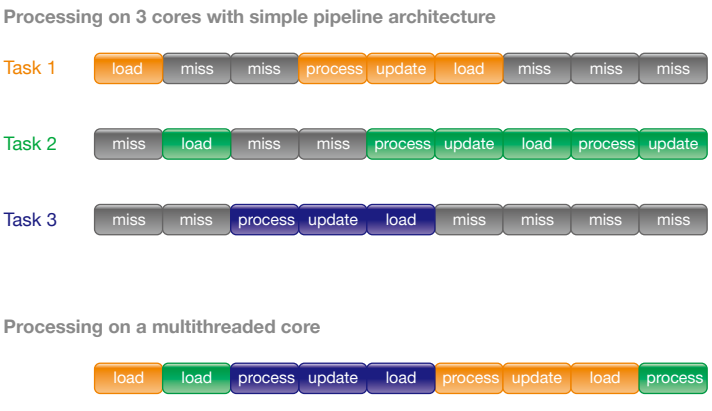


図 17: 複数のタスクの擬似同時プロセスにより、パイプライン上でのストールを回避

図 17 は、3 つのタスクを単純なパイプラインアーキテクチャの 3 つのコアで処理した場合（上側）と、マルチスレッドコアで実行した場合（下側）を図示したものです。

ハードウェアによるマルチスレッド化が機能するには、タスクを高速切り替えできる必要があります。これは、各タスクにそのコンテキスト用の専用レジスタセットを割り当てることで可能になります。この手法では、コアが提供するレジスタセットの個数から、並列で処理できるタスク数を簡単に推定できます。MIPS 34K の場合、レジスタセット数は 5 つになります。この方法では、1 つの「実行ユニット」（いわゆるハードウェアスレッド）が各レジスタセットから得られます（図 18 を参照）。

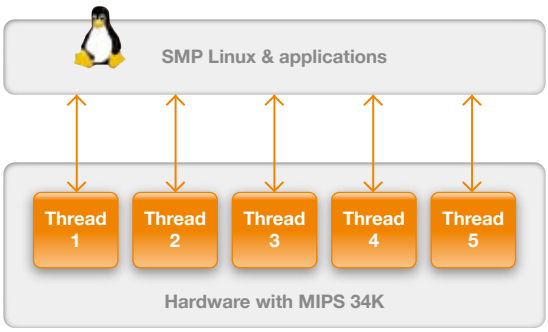


図 18: マルチスレッドコア上で SMP Linux(ここでは MIPS 34K)

SMP オペレーティングシステム

ハードウェアレベルでは、プロセスの並列化は対称型マルチコアプロセッサまたはマルチスレッドコアにより実装します。ここで、並列化を管理するためのソフトウェアが必要です。これには通常、オペレーティングシステムを使用します。

その一種に、同一実行ユニットを使用するハードウェアに特化している、対称型マルチプロセッシング (SMP) を実装するオペレーティングシステムがあります。SMP オペレーティングシステムの主な特徴として、プログラム実行時に、使用可能なコアまたはハードウェアスレッドにタスク / プロセスを動的に分散させる機能があります。

その他の特徴 :

- オペレーティングシステムの 1 つのインスタンスで、すべてのコアまたはハードウェアスレッドを処理します。
- 各アプリケーションプロセスを、各コアまたはハードウェアスレッド上で実行できます。通常、1 つのコアまたはハードウェアスレッドに厳密に割り当てられるのはカーネルだけです。
- すべてのコアまたはハードウェアスレッドで、リソース (メモリ、外部インタフェース、外部デバイスなど) の要求 / 使用権限が同等になります。
- オペレーティングシステムがリソースをコアまたはハードウェアスレッドに分散する機能を提供します。

新しいデバッグコンセプト

現時点の TRACE32 コンセプトでは、1 つのデバッガインスタンス (Core View) でデバッグできるコアは 1 つだけです。これまでは、マルチコアプロセッサも非対称型マルチプロセッシング (AMP) システムとして動作していたため、このコンセプトはマルチコアプロセッサでも問題なく機能していました。非対称型マルチプロセッシングでは、オペレーティングシステムの独立インスタンスが各コア上で実行されます。AMP システムの場合、どのコアでどのプロセスを実行するかが常に静的に決定されます。この方法では、デバッグ情報も対応するコアに一意に割り当てることができます。

これとは対照的に、SMP システムでは、プロセスは実行時までコアまたはハードウェアスレッドに割り当てられません。このため、選択したコアまたはハードウェアスレッドのデバッグ専用のデバッグインスタンスを開始しても意味がありません。

System View

AMP システムでは、問題なく機能する Core View の代わりに、SMP システム対応の System View が用意されました。

System View では、TRACE32 デバッガのインスタンス 1 つだけですべてのコアまたはハードウェアスレッドをデバッグします (図 19 を参照)。System View でも、1 つのコアまたはハードウェアスレッドを元に情報が表示されます。ここでは、コマンドにより別のコアまたはハードウェアスレッドの情報に切り替えて表示することができます。

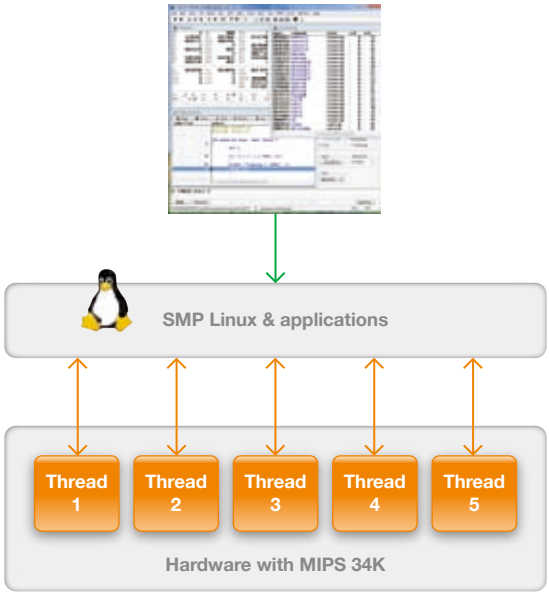


図 19:1 つの TRACE32 デバッガ GUI で全コア或いはスレッドのデバッグに対応

以下のような手順でプロセスをデバッグできます：

1. 実行中のタスク / プロセスの現在のリストを使用し

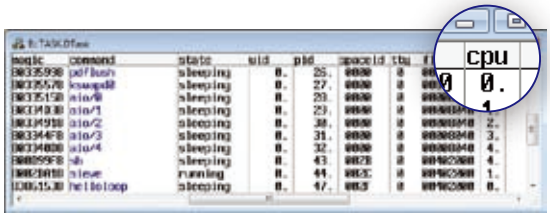


図 20:" TASK.DTASK" ウィンドウでは、SMP Linux がアサインしているコア或いはハードウェアスレッドを表示

て、タスク / プロセスを実行しているコアまたはハードウェアスレッドを特定します（図 20）。Linux では、コアまたはハードウェアスレッドではなく CPU という用語が使用されます。

2. コマンドを使用して、デバッガで表示する情報を目的のコアまたはハードウェアスレッドのものに切り替えます（図 21）。

共通ブレークポイント

SMP オペレーティングシステムでは、実行時までアプリケーションプロセスがコアまたはハードウェアスレッドに割り当てられません。このことは、オンチップブレークポイントの設定にも影響します。

簡単な例を紹介します。sieve プロセスが、変数 xyz に書き込むとすぐに停止する例です。sieve アプリケーションプロセスが実行されるコアまたはハードウェアスレッドをデバッガで事前に特定できないため、デバッガでこの要求を実装できるようにするには、このブレーク条件用にすべてのコアまたはハードウェアスレッドのブレークロジックをプログラムする必要があります。つまり、各コアまたはハードウェアスレッドに専用のブレークロジックがある場合でも、すべてのコアまたはハードウェアスレッドで共通のブレークロジックを共有しているように、デバッガでブレークポイントを設定します。

ソフトウェアブレークポイント（実装ではメモリ内の元の命令がブレーク命令で一時的に上書きされる）の設定時は、AMP システムからの変更はありません。オペレーティングシステムは、各プロセスのアドレス空間を相互に保護します。ただし、同一プロセスが複数回起動される場合は、たとえば Linux ではプログラムコードは 1 回しかロードされません。したがって、アプリケーションプロセスの各インスタンスが、同一のプログラムメモリやそこで設定されたブレークポイントを認識します。プログラム実行が指定のアプリケーションプロセスだけで停止したことを確認するため、TRACE32 デバッガではプロセス固有のソフトウェアブレークポイントを設定できます。

まとめ

ローターバッハは、TRACE32 コンセプトの計画的な拡張により、SMP オペレーティングシステムを使用して複数のコアまたはハードウェアスレッドを制御する組み込みシステム設計の簡単なデバッグを実現しました。この新しいコンセプトによる最新機能は、特定の CPU アーキテクチャ用デバッガとさらにそのデバッガで提供されるデバッグプローブにマルチコアライセンスなどの追加 / 拡張ライセンスが組み込まれていれば、ご使用になれます。

MIPS 34K のサポートに続いて、2008 年初めには ARM および PowerPC アーキテクチャ用 SMP システムのデバッグのサポートも計画されています。

図 21:TRACE32 ウィンドウではコア或いはハードウェアスレッドのコンテクストだけを表示．ステータスライン上にコア或いはハードウェアスレッドの番号を表示．

RTOS サポート機能の充実

ARM および SH4 用 Windows CE 6.0

TRACE32 で、ARM および SH4 アーキテクチャ用の Windows Embedded CE 6.0 もサポートされました。

2006 年末頃から、開発者は Windows CE の新バージョンをプロジェクトに使用できるようになりました。バージョン 6.0 では、カーネルアーキテクチャに根本的な変更がいくつか加えられました。たとえば、2GB のプロセスアドレス空間がサポートされました。バージョン 5.0 では、プロセスアドレス空間は 32MB に制限されていました。

2007 年後半以降、TRACE32 Windows CE Awareness バージョンが更新され、TRACE32 デバッガを使用して、すべてのプロセス、スレッド、ライブラリを同時に監視 / テストできるようになりました。また、リアルタイムトレースツールと組み合わせることで、Windows CE スレッドの実行時解析も可能になりました。さらに、Windows CE Awareness の設定が簡単になりました :

- デバッガでカーネルオブジェクトの自動検出がサポートされました。これにより、カーネルのシンボル情報を明示的にロードする必要がなくなりました。
- デバッガの Windows CE Autoloader により、プロセスおよびライブラリのシンボル管理が簡単になりました。Autoloader が設定されていれば (sYmbol.AutoLOAD.CHECKWINCE)、必要なときに対応するシンボル情報が自動的にデバッガにロードされます。

Windows CE Platform Builder との統合

2007 年初めに、ローターバッハのデバッガを Windows CE Platform Builder のハードウェアデバッグバックエンドとして使用できるようになりました。

ローターバッハは、Windows CE のカーネルおよびアプリケーションプロセスのデバッグ用に、ローターバッハのデバッガを Windows CE Platform Builder の内部デバッガ経由で制御できる eXDI2 ドライバを提供しています。

ターゲットデバイス接続を使用して、まず TRACE32 eXDI2 ドライバを Windows CE Platform Builder (PB) に統合します。統合後は、PB のデバッガのすべてのデバッグコマンドが TRACE32 コマンドにバックグラウンドで変換され、ローターバッハのデバッガに引き渡されます。

eXDI2 ドライバは、Windows CE 5.0/6.0 および Windows Mobile Versions 5/6 で利用できます。

Extensions

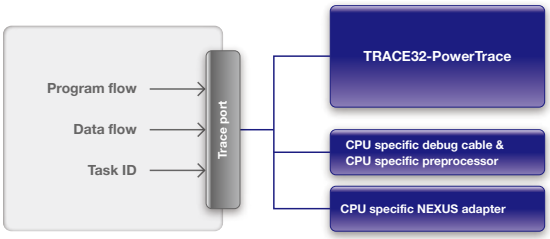
- Linux - Xenomai スレッドのサポート
- Linux - シンボルオートローダー
- OSE - 5.2 のロードモジュールのサポート
- PXROS - C167 用に再設計

新サポート RTOS

DSP/BIOS for TMS320C28xx	Available	RX4000 for MIPS	Available
eCOS for MIPS	Available	ThreadX for ARC and Blackfin	Available
FreeRTOS for ARM and MicroBlaze	Available	ThreadX for Xtensa	Planned
HI7000 for SH4	Available	Windows CE 6.0 for ARM and SH4	Available
Linux for ARC and MicroBlaze	Planned	µClinux for Blackfin	Planned
LynxOS-SE for PowerPC	Planned	µClinux for MicroBlaze	Available
NetBSD for ARM and PowerPC	Available	µC/OS-II for V850	Available
OS/9 for ARM and PowerPC	Available	µC/OS-II for Xtensa	Planned

LAUTERBACH

新しいプリプロセッサ /NEXUS アダプタ



現在の多くのシステムオンチップ（SoC）設計では、実行時情報をチップ外部で確認できるトレースポートを利用しています。そのような情報には、プログラムが実行する命令やデータ転送があります。オペレーティングシステムを使用している場合は、現在のタスク ID の出力が役立ちます。トレースポートとして、ARM アーキテクチャの ETM および PowerPC アーキテクチャの NEXUS がよく知られています。

ローターバッハは、実行時情報を記録するためのリアルタイムトレースツール TRACE32-PowerTrace を提供しています。ハードウェアレベルでは、アーキテクチャ固有のプリプロセッサまたは NEXUS アダプタがトレースポートからデータを受信し、PowerTrace ハードウェアのトレースメモリ（最大 4GB）に転送します。この記録情報をもとに、効率的なトラブルシューティングや設計に対する総合的な実行時解析を行うことができます。

ローターバッハは現在、30 を超えるアーキテクチャ向けにプリプロセッサや NEXUS アダプタを提供しています。2008 年にはさらにサポート対象が拡大する予定です。

MPC5500 アーキテクチャ用の新しい NEXUS アダプタ

2008 年 3 月以降、最新技術を採用した MPC5500 アーキテクチャの NEXUS アダプタのバージョンが利用可能になります。新種類の NEXUS ポートに柔軟かつすぐに対応できるように、このアダプタは最大 16 ビット MDO のポート幅および 1.0 ～ 5.0V の電圧範囲をサポートしています。

ローターバッハは、特定のプロセッサの NEXUS ポートに対応させるためのコンバータを提供しています。

高速で流れるトレースポート上のデータに対し、最適なトレース信号のサンプリングを保証するため、新しい NEXUS アダプタには、最適なサンプリングインスタントを自動的に設定する FPGA（AUTOFOCUS 技術）も採用しています。

MPC5500 AF NEXUS アダプタ

- NEXUS ポート幅 最大 16-bit MDO、2-bit MSEO、EVTI、EVTO
- 最大 200 MBit/s データ転送速度
- SDR：200 MHz または DDR：100 MHz
- 対応電圧：1.0 V ～ 5.0 V
- 複数の NEXUS ポートコネクタ用のアダプタ
- AUTOFOCUS テクノロジ

新しい AUTOFOCUS II プリプロセッサ

プリプロセッサで 500MHz 以上のトレースポート周波数を処理するため、2006 年に AUTOFOCUS II 技術が開発されました。この技術の最も重要な機能は、トレースデータに最適なサンプリングを自動的に設定する機能です。現在では、この技術をもとに、一連の DSP 向けのプリプロセッサがさらにたくさん提供されています。

新 AUTOFOCUS II プリプロセッサ

Preprocessor for Ceva-X	Q1/2008
Preprocessor for StarCore	Q3/2008
Preprocessor for TMS320C55x and TMS320C64x	Q1/2008

MicroBlaze

2007 年 11 月以降、MicroBlaze アーキテクチャ向けプリプロセッサも提供されています。MicroBlaze Trace Core では、22 ピン幅のトレースポートを採用し、プログラムおよびデータフロー（オプション）をオフチップで確認できます。100MHz を超えるトレースポート周波数をプリプロセッサで処理できるようになりました。

CombiProbe

実行時の重要なシステム情報を高速で出力するため、一部のコアは単純なトレースポートを採用しています。これらのポートでアプリケーションプログラムを利用して、ユーザー定義データをオフチップで確認できます。ローターバッハは、このデータの記録および処理用に CombiProbe という新ツールオプションを提供しています。以下では、この新製品のハードウェアおよび動作について説明します。

インスツルメンテーション

ほとんどの開発者は、最も効率的な方法として、単純な printf() を実装したようなテストシナリオに慣れています。典型的な例は、診断情報の出力やシステムイベントのログがあり、このようなシナリオでは、アプリケーションがすでに想定通り動作していて、次に運用テストを実行する必要があるとされています。通常、この動作テストにはコード計測を使用します。

プログラム実行時の重要なシステム情報をインスツルメンテーションを使って表示するには、以下の手順を実行する必要があります :

1. 必要な情報を提供する命令をアプリケーションプログラムに挿入します。
2. この情報を表示する方法を決定します。通常、これには RS-232 や Ethernet などの外部通信インタフェースを使用します。

インスツルメンテーションには、テスターが必要な情報だけを確認できるという大きな利点があります。ただし同時に、原則的にオペレーティングシステムで追加の通信インタフェースを処理する必要があるため、大幅な通信オーバーヘッドが生じます。インタフェースが低速の場合、実際のアプリケーションでかなりの数の実行時問題が発生するおそれがあります。

システムトレース

アプリケーションプログラムが生成するシステム情報の出力専用のトレースポートをコアが提供している場合、計測時の通信オーバーヘッドの大部分を削減できます。ARM 社は、IP プロバイダとして、同社の CoreSight 技術の一環としてこのようなソリューションを提供しています。

CoreSight Instrumentation Trace Macrocell (ITM) は、基本的に以下のように動作します :

1. アプリケーションプログラムが、ITM に割り当てられた 32 ビットメモリマップドレジスタに、確認が必要な情報を書き込みます。
2. ITM はこの情報を、シリアルワイヤー出力 (図 22 を参照) 経由で直接、または CoreSight Trace Port Interface Unit (TPIU) 経由で他のトレースデータと併せて提供します。

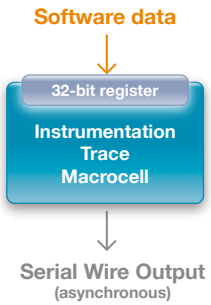


図 22: シリアルワイヤ出力による CoreSight インストロメンテーショントレース

多くの携帯電話メーカーは、自社製チップで同様の独自ソリューションをすでに採用しています。しかし、コスト削減のため、このようなトレースポート用の統一規格に関心が集まっています。2007 年 5 月、MIPI Alliance の Test and Debug ワークグループは、4 ビットトレースポートと 34 ピンのデバッグおよびトレースコネクタ用規格として「System Trace」を策定しました。

規格化に併せてトレースポートの機能が拡張されました。System Trace では、アプリケーションプログラムが生成するシステム情報とは別に、ハードウェア情報も出力できます (図 23 を参照)。たとえば、プログラマブルバスウォッチャーは、特定のバスサイクルを検出して表示できます。また、信号モニターは、選択したチップ内信号の状態を返すことができます。

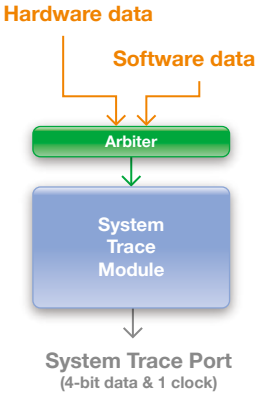


図 23:MIPI アライアンスによるシステムトレース

CombiProbe ハードウェア

ローターバッハは、Test and Debug ワークグループのメンバーとして、System Trace の仕様策定に積極的に参加しました。同時に、「CombiProbe」というそれに適したデバッグ / トレースオプションの開発にも携わりました。2007 年 10 月から、ローターバッハではこの新製品を販売しています。

CombiProbe は、特殊なデバッグケーブルと 128MB のトレースメモリを組み合わせた製品です。CombiProbe は、POWER DEBUG INTERFACE / USB 2 などのローターバッハの汎用ハードウェアに接続できます (図 24 を参照)。ターゲットシステム接続には、MIPI 規格の 34 ピンのデバッグ / トレースプラグを使用します。このプラグは、JTAG ポート用コネクタと 4 ビットの System Trace ポートで構成されます。

ローターバッハは、他のターゲットシステムコネクタ用アダプタも提供しています。

動作コンセプト

CombiProbe は、従来のデバッグ機能に加えて、新しい設定コマンドおよび記録したトレース情報の表示と解析用コマンドを提供します。

図 24:CombiProbe を取り付けた POWER DEBUG USB2

設定コマンドは、以下のとおりです :

- ITM.<subcommand> : CoreSight 技術の ITM を設定（図 25 を参照）
- STM.<subcommand> : System Trace モジュールを設定
- SystemTrace.<subcommand> : CombiProbe を設定

SystemTrace コマンドは、記録したトレースデータの解析および処理にも使用します。

トレースポートで表示されるシステム情報は、通常と同様に CombiProbe のトレースメモリに記録し、TRACE32 コマンドを使用して解析することができます。TRACE32 でいわゆるプロトコル API を使用し、生のトレースデータをアプリケーションでフォーマットすることで、システム情報を直観的に解析できます。

ただし、記録時間は CombiProbe のトレースメモリサイズによって制限されます。

記録時間を長くするには、CombiProbe の PIPE モードを使用します。PIPE モードでは、トレースデータがホストに直接転送されます。このモードで、CombiProbe のトレースメモリはバッファのみに使用されます。

ホストでは、トレースデータを直接ファイルに出力するように TRACE32 を設定できます。

記録後に、外部アプリケーションでシステム情報をフォーマットして解析することも可能です。

あるいは、外部解析ソフトウェアで TRACE32 FDX API を使用してトレースデータを収集し、記録中に処理することもできます（図 26 を参照）。

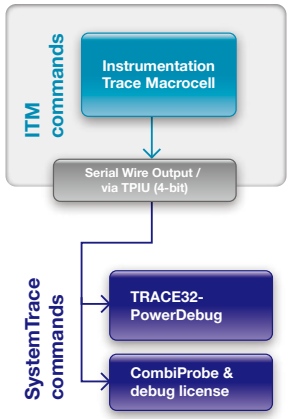


図 25: インストロメンテーション・トレース・マクロセルの設定に使用される ITM コマンド . システムトレースコマンドは計測したトレース情報の表示、解析に使用 .

まとめ

CombiProbe は、幅広いローターバッハ製品で実行時情報を記録するための新しいツールオプションです。複雑なテストケースについて、CombiProbe を TRACE32-PowerTrace と組み合わせて使用することもできます。すべてのローターバッハ製ツールは時間ベースが共通のため、コアのプログラム / データフローとともに重要なシステム情報を入力順に記録できます。

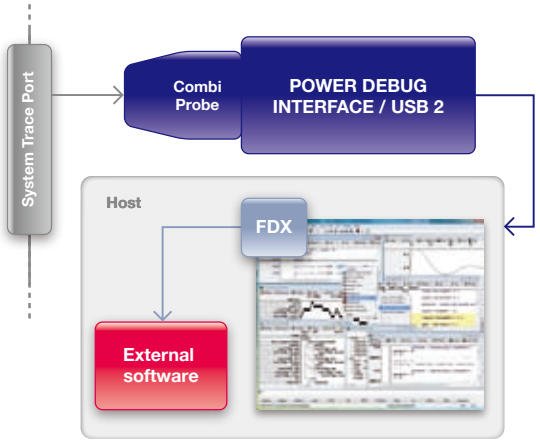


図 26: 外部ソフトウェアがトレース情報を処理できるように設定することも可能 .

3G/DigRF レポート

ローターバッハは、2007 年夏期から PowerIntegrator 用の新しいプローブを提供しています。このプローブを使用して、3G/DigRF インタフェースからのデータトラフィックを記録することができます。

記録では、以下のサンプリングレートが使用されます :

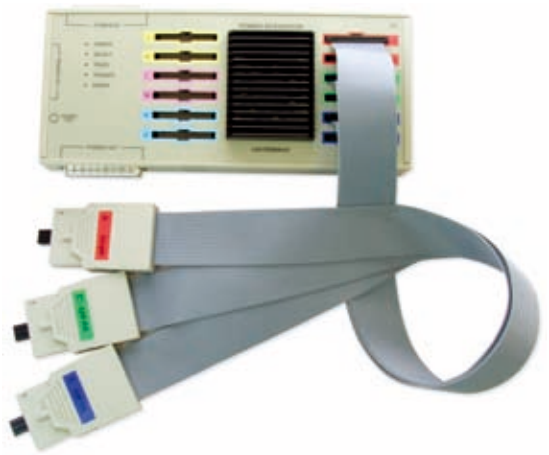
- 2GigaSample/s（RX/TX ラインの場合）
- 250MegaSample/s（SysClkEnable ラインの場合）

ラインのデータトラフィックにより、全負荷時で 25.156 ミリ秒～数時間の記録が可能です。高感度プローブと、非常に低負荷な入力信号でエラーのない記録が保証されます。

測定データは、プロトコルレベルで評価・解析したり、プログラムフローと時間相関することができます。

CombiProbe data

- デバッグケーブルと
  128MByte トレースメモリ
- ARM コア用 JTAG サポート
  - 標準的な JTAG
  - シリアル・ワイヤ・デバッグポート
  - cJTAG（IEEE P1149.7）、対応予定
- その他のアーキテクチャ用の
  標準 JTAG 対応
- トレースポート：
  - MIPI システムトレース
  - ITM シリアル・ワイヤ出力
  - ITM 4-bit CoreSight TPIU
  - コンティニュアスモード 4-Bit ETMv3
    対応予定
- 1 チャンネル - 200 MBit/s 帯域幅 I
  最大 4 トレースチャンネル
- 対応電圧： 0.3 V ～ 3.3 V、
  （許容電圧 ： 5 V）
- 34-pin ハーフサイズコネクタ
- 10 および 20 ピンハーフサイズコネクタ



ご意見をお寄せください

追加情報をご要望の場合は、下記アドレスまで電子メールでご連絡ください。

info@lauterbach.co.jp