リアルタイムデバッグ対応組み込み用ポータブル BIOS

Portable Embedded BIOS for Real-time Debugging

籠屋 健 [1)]
Takeru KOMORIYA

1) 東北大学大学院 情報科学研究科 システム情報科学専攻
(〒980-8579 宮城県仙台市荒巻字青葉 01　E-mail: komoriya @ paken.org)

ABSTRACT. A novel debug method for realtime embedded system is proposed. In case of debugging realtime process, usual 'breakpoint' may destroy the process, while proposed method of 'checkpoint' doesn't. At the checkpoint, the debug interface stops user program and communicates with host's debugger in very short time. 'REDMON(Realtime Enhanced Debug Monitor)' is developped for the target system, and 'GDB/RT(RealTime)' is used as realtime debugger. The system is examined on the robot control system.

1 背景

組み込み機器の開発では，ホスト PC で開発したプログラムをターゲットに転送して実行させる「クロス開発方式」を用いることが多いが，この場合，以下の 3 つの手順を繰り返してプログラムを開発することになる．

- ホスト PC 上でソースコードを記述し，ビルドする
- バイナリプログラムをターゲットに転送する
- プログラムを実行して動作を検証する（リモートデバッグ）

開発効率の向上のためには，このサイクルをできるだけ速く繰り返せるようにすることが一つのポイントである．

通常のアプリケーションの開発では，デバッガを用いて動作中のプログラムの状態を監視・変更することで，デバッグ効率を高めることが可能であるが，クロス開発で用いられるリモートデバッガでは，メカトロニクスや通信等の分野で必要な「リアルタイム処理」には対応できない．

すなわち，リモートデバッガを用いた一般的な操作では，一旦プログラムを停止 v させて変数の値等を監視するが，例えば周期的な処理によってモータにフィードバック制御をかけるようなプログラムをデバッグしようとする場合，プログラムを停止させている間にモータが回転してしまい，その後の制御が正しく動作しなくなってしまう．

このため，リアルタイム処理に関するデバッグを行うためには，プログラマ自身がデバッグ用のコードを記述してプログラムのビルドと転送を繰り返す必要があり，開発効率を大きく悪化させる要因となっている．

以上のことから，従来のクロスデバッガの操作性を損なわない形で，リアルタイムにデバッグを行うことのできるシステムが実現できれば，開発効率の向上に寄与すると考えた．

2 目的

本プロジェクトでは次のようなリアルタイムデバッグシステムを開発することを目的とした．

- リアルタイムなデバッグを実現するために，従来の「ブレークポイント」に加え「チェックポイント」という仕組みを導入し，プログラムを見かけ上停止させずにデバッグができるようにする．
- 上記のリアルタイムデバッグ機能を持ったモニタプログラムを開発する．シリアルポートや USB を経由したデバッグ機能に加え，簡単なブートローダ等を備える．さらに異なるアーキテクチャ・ハードウェア構成に対して移植性を高める工夫をする．
- ホスト側のデバッガはフリーソフトウェアである GDB を拡張する形で開発を行い，Linux, Windows 上で動作するマルチプラットホームとする．また，GUI を備えて容易に操作ができるようにする．

また，開発したシステムの効果を確認するため，実際にロボットを動かすアプリケーションを作成して動作を検証することとした．

3 リアルタイムデバッグの仕組み

通常のデバッガでは，予め指定した「ブレークポイント」でプログラムの実行を一時中断して変数や式の値等を調べたあと，ユーザの指示によりプログラムの実行を再開する．このような，デバッグ時にプログラムを長時間停止させる方法は，前にも述べたようにリアルタイムで実行される処理を破綻させてしまう可能性がある．

この問題に対処するため，ここでは従来のブレークポイントに加えて「チェックポイント」を仕組みを提案する．チェックポイントは，ブレークポイントと同様にユーザプログラムを一旦停止させるが，デバッグに必要なデータをやりとりしたあと，速やかにプログラムの実行を再開させる．

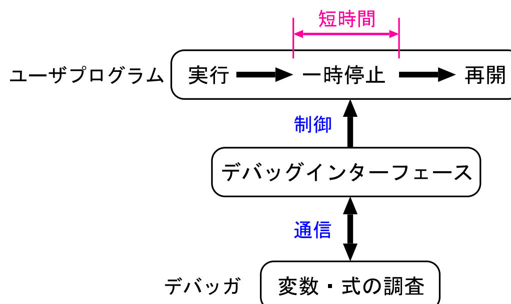

図 1: リアルタイムデバッグの仕組み

チェックポイントでの処理（変数の値の取得など）はチェックポイントの登録時に予めユーザが指定しておく．

この方式は，いわば余っているリソース (CPU および通信) を使用する．近年，組み込み用途でも高性能な CPU や高速な通信路が用いられるようになってきたが，それでも実際に使用する場合はデバッグ処理が消費するリソースをできるだけ小さく抑える必要がある．

4　リアルタイムデバッグモニタ REDMON

REDMON(Realtime Enhanced Debug MONitor) はターゲットとなるマイコンボード上で動作するデバッグモニタである．

（1）機能概要

REDMON はリアルタイムデバッグを行うためのインターフェースの他，ブートローダ機能，BIOS 機能，簡易モニタ機能，通信用デバイスドライバ機能を備えるものとした．

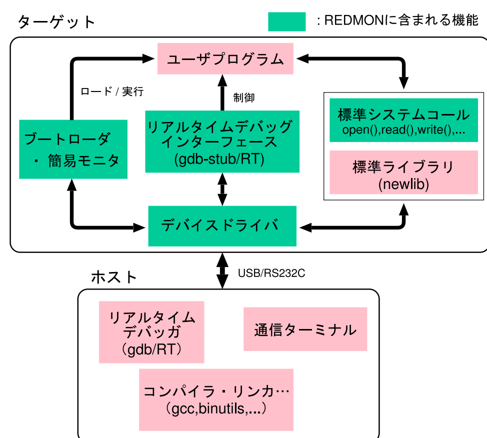

図 2: REDMON の機能

a）デバッグインターフェース (gdb-stub/RT)

通信ポート経由でホスト上のデバッガと通信し，ユーザプログラムの制御を行う．GDB のリモートデバッグプロトコルにリアルタイムデバッグを行うための機能を追加したデバッグインターフェース (gdb-stub/RT) を実装する．

b）ブートローダ・簡易モニタ

通信ポート経由で Motorola S-record 形式のファイルを受信し，RAM に書き込む機能を備える．

また，ユーザがシリアルターミナル経由でメモリ内容のダンプやプログラムの実行等の制御を行うための簡易モニタとしての機能も備える．

c）BIOS 機能・システムコール

ユーザプログラムから，通信デバイス制御・割込制御・キャッシュ制御・電力制御の各 BIOS 機能を備え，組み込み用標準ライブラリである Newlib から利用可能な基本的なシステムコールを備える．これにより，標準ライブラリを用いて周辺デバイスを扱うことができるようにする．

（2）実装

a）使用言語

REDMON は C 言語およびアセンブリ言語を用いて記述した．ただし移植性を考慮して，スタートアップルーチンと特殊な関数以外はすべて C 言語で記述した．

b）対応アーキテクチャ

対応するターゲットのアーキテクチャは SH4,SH2,H8/300 の 3 種類とした．

c）Newlib とのインターフェース

標準ライブラリである Newlib がシステムコールを呼び出そうとすると，ソフトウェア割り込みが発生する．REDMON はこの割り込みを捕らえることでシステムコールを実行する．

open(), close(), read(), write(), exit(), sbrk(), wait() が実装されている．

d）デバイスファイル

シリアルポートや USB ポートは，ユーザプログラムからはデバイスファイルとしてアクセス可能になっている．

デバイスファイルは

```
fd = open("SCI:", O_RDWR|O_NONBLOCK);
```

のように名前を指定して利用することができる．利用可能なデバイスファイル名には以下のようなものがある．

`STDIN` ： **標準入力**
`STDOUT`： **標準出力**
`SCIF` ： `FIFO` **付きシリアルインターフェース**
`SCI` ： **シリアルインターフェース**
`USB` ： `USBN9603`

e）BIOS コール

ユーザプログラムから通信デバイス制御・割込制御・キャッシュ制御・電力制御を簡単に行うために，BIOS コールが実装されている．BIOS コールはソフトウェア割り込みで実装されているが，ユーザから利用しやすいように以下のような wrapper 関数が用意されている．

f）デバッグインターフェース (gdb-stub/RT)

リアルタイムデバッグインターフェースである gdb-stub/RT は，独自のデバッグプロトコルに従ってデバッガとの通信とユーザプログラムの制御を行う．詳細については次章で述べる．

5　リアルタイムデバッガ GDB/RT

GDB/RT はホスト上で動作するデバッガであり，フリーソフトウェアである GDB を拡張してリアルタイムデバッグ機能を追加することで実現した．GUI を備え，Linux および Windows 上で動作する．

（1）設計

a）リアルタイムデバッグ用コマンドの追加

リアルタイムデバッグを実現するために，従来のブレークポイントに加えてチェックポイントという仕組みを導入する．

従来のブレークポイントを使ってリアルタイムデバッグのようなことをしようとした場合，以下の手順を自動的に行うことになる．

1) ブレークポイントによるユーザプログラムの停止
2) 指定した変数の読み込み
3) ユーザプログラム実行を再開

この場合，ターゲットとホスト間でパケットの送受信を何度も繰り返すことになり，通信をするだけで非常に長い時間がかかってしまう．

このため，ストアドプロシージャと呼ぶ方式を用いて，チェックポイントで実行される手続きを予めターゲット側に記憶させておき，チェックポイントでの通信回数を最大でも往復 1 回に抑えた．

ストアドプロシージャを実現するために，GDB のリモートデバッグコマンドに以下のものを追加した．

Set Check Point (C) :
指定したアドレスにチェックポイントを設定する．

Start Procedure (p) :
直前に設定したチェックポイントに対し，コマンドリストの設定を開始する．これ以降に与えられるコマンドは

チェックポイントに対するコマンドリストとして登録される．

End Procedure (e)：
コマンドリストの設定を終了する．

Delete Check Point (D)：
指定したアドレスのチェックポイントを削除する．

b）リモートシリアルプロトコルの改良

GDB でリモートデバッグ時に用いられるリモートシリアルプロトコルは，単純で可読性が高いかわりにデータが冗長であり，高速なレスポンスが必要となるリアルタイムデバックには適さない．

このため冗長性を下げたパケット構造を採用した．

内容	コマンド	データ長	データ	チェックサム
バイト長	1	1	0–255	1

（2）実装

GDB-5.2 を拡張する形で実装を行った．C 言語で記述されている．

リアルタイムデバッグ時のデータの入出力は，コンソールの他にデバイスファイルを経由して行えるように実装した．これによって変数の値をリアルタイムでグラフ描画ソフトに渡したり，後述するシミュレータとの接続に利用することができる．

なお，GUI 機能は Insight を用いて実現した．

（3）GDB/RT の操作

GDB に元々備わっている機能についての説明は省き，ここでは新たに追加したリアルタイムデバッグ機能の使い方について説明する．

以下の操作はコンソールウィンドウ上で行う．

a）チェックポイントの追加

check コマンドを用いると指定した場所にチェックポイントを追加する．行番号の代りに関数名を指定すると，その関数のエントリにチェックポイントを設定する．ファイル名は省略可能である．

check ［**ファイル名：行番号**］
check ［**ファイル名：関数名**］

check コマンドに続けて，設定したチェックポイントで実行するコマンドリストを入力する．

output ［**変数名**］: **指定した変数を出力**
input ［**変数名**］ : **指定した変数にデバイスファイルに書き込まれた値を設定**
echo ［**テキスト**］: **指定したテキストを表示**
end : **コマンドリストの終了**

b）チェックポイントの表示・削除

引数なしで check コマンドを実行すると，登録されているチェックポイントの一覧を表示する．

check

チェックポイントを削除するには clear コマンドを用いる．

clear ［**ファイル名：行番号**］
clear ［**ファイル名：関数名**］

c）デバッグ用デバイスファイルの利用

チェックポイントでのデータは，コンソールウィンドウの他に，デバイスファイル (/dev/rtsim) にも出力される．

このデバイスファイルを利用することで，データをファイルに保存したり，グラフ描画ソフトにデータを渡すことが容易にできる．

また，逆にデバイスファイルにデータを入力することもできる．チェックポイントのスクリプトに input コマンドを記述すると，/dev/rtsim に書き込まれたデータが指定した変数に入力される．この機能は主にシミュレータとの接続に用いる．

6 リアルタイムデバッグ用 USB ドライバ

USB-RDID(USB Realtime Debug Interface Driver) は，Linux 上で動作する USB1.1 用のデバイスドライバである．USB 経由でホストとターゲット間の通信を行うとともに，ユーザプログラムとデバッガが通信路を共有する場合に，データの送り先を適切に振り分ける役目を担う．

（1）設計

キャラクタ型デバイスファイルを通して，USB 経由でターゲットと通信するものとした．送信・受信それぞれに USB のエンドポイントを一つ使用し，バルク転送を行う．

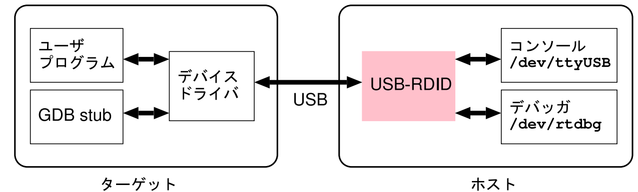

図 3: USB-RDID の機能

ユーザデータとデバッグデータを区別するために，パケットに識別子を設けて，それぞれを別々のデバイスファイルにアサインすることにした．

なおパケット長は USB のエンドポイントのパケットサイズ (64byte) に収まるようにして，ユーザデータとデバッグデータが混在する場合でも，遅延を最小限に抑えられるようにしている．

ヘッダ		データ
ID(1byte)	パケット長 (1byte)	データ (1–62byte)

(ID=0:ユーザデータ, ID=1:デバッグデータ)

（2）実装

Linux Kernel 2.4 系用のカーネルモジュールとして，C 言語で実装した．

下図のように，USB Serial Driver の下部モジュールとして機能し，シリアルドライバと互換性を保つようにした．このため minicom 等の通常のシリアルターミナルプログラムを用いてアクセスが可能である．

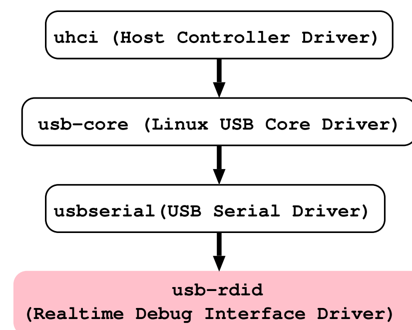

図 4: USB ドライバの依存関係

なお，デバイスがホスト側からオープンされている状態でターゲットとの接続が切断された場合，通常はターミナルプログラムの再接続が必要となる．このため，デバッグ時など頻繁にターゲットをリセットするような使い方では余分な操作が増えてしまう．

そこで，USB-RDID ではターゲットが切断されても，見掛け上，ホスト側のプログラムからは切断されていないように見えるようにした．この場合，ターゲットが再起動すれば，ホスト側はデバイスを再オープンせずにそのまま通信が可能である．

なお USB-RDID を用いた場合のデータ転送速度は，実測で 4Mbps 程度が得られている．

（3）使用方法

USB Serial Driver が適切にロードされている状態で, 以下のようにカーネルモジュールとしてロードして用いる．

```
$ insmod usb-rdid.o [オプション]
```

オプションには以下のものがある．

vendor=VID : USB のベンダ ID を指定
product_base=PID : USB のプロダクト ID を指定
debug=1 : デバッグモード有効
buffer_size=SIZE : バッファサイズを指定

USB 機器が接続されると，キャラクタデバイス /dev/ttyUSB* および/dev/rtdbg* が割り当てられる．前者はユーザプログラムの通信路として用いられ，後者はデバッグデータをやりとりするために用いられる．

7 動作検証

（1）ロボット制御の例

本プロジェクトで開発したシステムの動作を検証するため，実際にロボットを制御しながらリアルタイムデバッグを行う実験を行った．

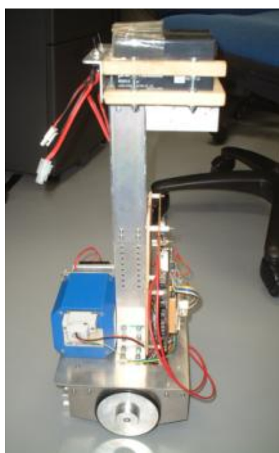

図 5: 倒立振子型ロボット

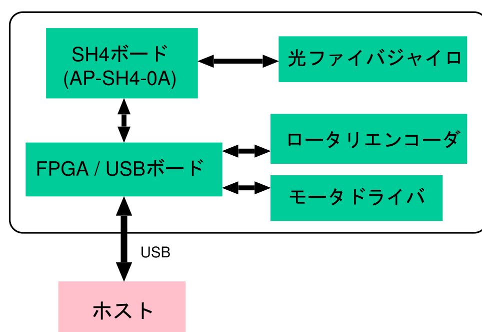

図 6: ロボットのブロック図

図のような倒立振子型ロボット (二輪のみで安定に立つように制御される) を用い，REDMON を使ったユーザアプリケーションが正常に動作し，周期的に実行される制御ルーチン内の変数をリアルタイムに監視できることを確認した．

（2）ロボットシミュレータとの接続

リアルタイムデバッガ経由で，ターゲット上で実行されている制御プログラムと，ロボットシミュレータを接続する実験を行った．

前出の倒立振子型ロボットをシミュレートするため，力学シミュレータライブラリである ODE(Open Dynamics Engine) を用いてシミュレータを開発した．

ターゲットボードとシミュレータとは，リアルタイムデバッガの入出力デバイスファイル (/dev/rtsim) を経由して通信を行う．ターゲットではシミュレータから渡される車輪の回転数と車体の角速度を基にしてモータへの制御出力を決定し，結果をシミュレータに渡している．

シミュレータでの演算量が大きいため，制御周期を現実の値よりも大きくする必要があったが，シミュレータは正常に動作し，また同時にリアルタイムデバッグが行えることを確認した．

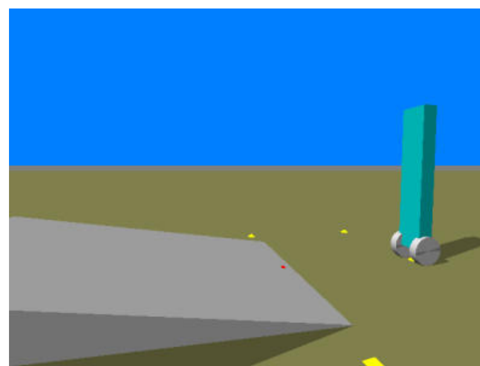

図 7: ロボットシミュレータ

（3）リアルタイム性に関する検討

実験に用いたロボットの制御周期は 10ms であり，ターゲット上のプログラムはこの周期で正確に動作したが，リアルタイムデバッグ時にときどきデータを取りこぼすという問題が発生した．

この原因を調査したところ，ホスト側のデバッガの応答が間に合わない場合があることが分かった．当初はホスト PC はターゲットに比較して十分に高速であるという仮定をおいていたが，実際にはホスト側ではリアルタイム性は保証されず，数十 ms の遅延が発生した．この遅延はホスト側でデバッガ以外のプロセスが多く実行されるほど顕著になる．

対策としては，デバッガ以外のプロセスをできるだけ実行しないことや，OS のスケジューラの周期を変更する方法などがあるが，根本的に解決するためにはホスト側のデバッグプロセスをリアルタイム化する必要がある．

8 今後の課題と展望

（1）ホスト側デバッグ処理のリアルタイム化

全章で述べたように，安定・高速なリアルタイムデバッグを実現するためには，ターゲット側のみならずホスト側のデバッグ処理のリアルタイム化が重要であることが判明した．特に実機の機能の一部をシミュレータで代替するような場合は，完全にリアルタイム性を保証することが必要である．

RT-Linux や ART-Linux のように通常の OS をリアルタイム拡張した実装もいくつか存在し，このような OS を用いると指定したプロセス（またはスレッド）のみをリアルタイム化できるため，デスクトップ OS としての機能を損なうことなく，デバッグ処理だけをリアルタイム化できる可能性がある．

（2）幅広いアーキテクチャへの対応

ターゲット側のデバッグモニタは CPU のアーキテクチャ毎に用意しなければならない．本プロジェクトでは 3 種類の CPU に対応したものの，広く利用されるためには十分とは言えない．組み込み用のメジャーなアーキテクチャを広くカバーすることが望まれる．

（3）各種ネットワークへの対応

本プロジェクトでは，デバッグ用の通信路として従来から広く用いられていた RS-232C に加えて USB1.1 をサポートしたが，適用範囲をより広げるためにはより多くのデバイスをサポートする必要がある．

具体的には組み込み機器で広く用いられはじめた USB2.0，Ethernet や，自動車内のネットワークとして有力視されている CAN(Controller Area Network) 等が考えられる．

9　まとめ

本プロジェクトでは，組み込み分野でのプログラム開発の効率化を目指して，リアルタイムデバッグの新しい考え方を提案し，一定の条件下で実際に動作するシステムを構築した．

ターゲット用のデバッグモニタは複数のアーキテクチャに対応していて，標準ライブラリも利用できるようになっており，プログラミングの効率化に寄与するものと考えられる．

リアルタイムデバッグの仕組みは制御周期がある程度長ければ問題なく動作し，デバッグ作業の効率化を実現する．

以上，本プロジェクトのシステムを真に実用的なものとするためにはまだいくつかの課題が残されているものの，一定の成果は得られたと考える．

10　参加企業及び機関

本プロジェクトは IPA 平成 14 年度未踏ソフトウェア創造事業「未踏ユース」において，プロジェクト管理組織である株式会社 創夢とともに行われた．

本プロジェクトのプロジェクトマネージャである竹内郁雄教授，ならびに関係者の方々に謝意を表する．