リアルタイムカーネル勉強会用資料

gdb-stub

Eva A. Barcelon
豊橋技術科学大学 情報工学
cybil@ertl.ics.tut.ac.jp

1　概要

Stub は通常ターゲットの ROM 上に置かれて実行され，ホスト PC で動作する gdb と通信を行うことによりデバッグを可能にするプログラムである．stub を用いることにより、プログラムのダウンロード、ブレーク、ステップ実行、などが可能となる

デバッグ時に使用する breakpoint は，事前に設定することも，デバッグ中に設定することも可能である．事前に設定する breakpoint は，プログラマが明示的にプログラム中に記述する必要がある．breakpoint を配置したプログラムの実行が breakpoint によって中断されたら (以下 break という)、スタブが呼び出される。

SH 版の stub では、breakpoint を trapa 命令を用いて実現している。事前に設定する breakpoint は”trapa #0xff” で、デバッグ中で設定する breakpoint は” trapa#0x20” である。後者では、break したい命令が既にメモリ上にあり、これを trapa 命令で置き換えなければならない。また、置き換えられた命令と break する前の PC をもどさなければならない。このため、事前に設定する breakpoint とデバッグ中に設定する breakpoin を区別しなければならない。またステップ実行では”trapa #0x20” を用いる。

その他、プログラム実行中に”Ctrl-C” を押すと gdb は ̂C(’\003’) をシリアルポートに出力する。このシリアルポートの受信割り込みによってもスタブは実行される。このポートの受信割り込みは、優先順位が 15 に設定され、これにより、ユーザープログラムの実行を gdb から停止することが可能となる。なお stub は割り込みマスク 15 で実行されるので gdb との通信中割り込みがかかることはない。

また、ユーザープログラムから gdb へのログの出力を”trapa #0x40” で、割り込みの登録を”trapa #0x50” で行う。

trapa #0x3f を実行することにより stub の機能を呼び出し、gdb のコンソールへの出力や例外割り込みの登録ができる。

2　通信パケット

stub は gdb との通信を行う際、以下に示すパケットの方式を使う。

$packetinfo#checksum

通信パケットは文字’$’ で始まり、packetinfo(stub と gdb 間の通信で用いるコマンド) と checksum を’#’ で区切
サポートされているコマンドのいくつかを以下の表に示す。

表 2.1: パケット情報

gdb 側のコマンド	動作	stub からの返事
g	CPU レジスタの値を要求する	XXX...X 各レジスタデータ バイトは 2 桁の 16 進数で表 ターゲットでのレジスタの は GDB と同じ
GXX...XX	CPU レジスタの値を設定する	OK
mAA...AA,LLLL	AA...AA 番地から LLLL バイト数 を読み込む	XX...XX メモリの内容。 LLLL で指定したバイトの よりも少ない場合もある
MAA...AA,LLLL,XX...XXX	AA...AA 番地から LLLL バイト数だけ XX...XX を書込む	全て書き込み成功なら OK
cAA...AA	プログラムの実行再開。 AA...AA 番地を指定すれば、その番地から プログラムの実行を再開できる。	
sAA...AA	PC が示すアドレスからステップ実行を再開する。 AA...AA 番地を指定すれば、その番地から プログラムのステップ実行を再開できる。	
?	実行が中止になった理由をシグナルナンバー として要求	SAA AA はシグナルナン

3　構成

Stub は以下のモジュールから構成されている。

1. スタートアップルーチン (init_XXX.S)
2. setjmp,longjmp(setjmp.S)
3. 例外/割り込み処理モジュール (entry.S)
4. メインルーチン (sh-stub.c)

3.1　スタートアップルーチン

stub は単体で ROM からブートするため、周辺モジュールを初期化しなければならない。ターゲットによって、
期化が異なるため、init_card-e09a.S、init_dvesh3.S、init_rsh3.S とターゲット毎に用意されている。初期化が必要

モジュールは、クロックコントローラ、バスステートコントローラ、キャッシュコントローラ、割り込みコントローラ、gdb と通信するためのシリアルモジュールがある。また、通常のプログラムと同様スタック、ステータスレジスタ bss/data セクションの初期化も行う。全ての例外/割り込みを捕まえるため、VBR+0x100,0x400,0x600 番地に例外/割り込み処理ハンドラがくるようにベクタベースレジスタ (VBR) を設定する。

これらの初期化が終了すると、“trapa #0xff” を実行してメインルーチンに制御を移す。

3.2 setjmp,longjmp

setjmp はレジスタ R8〜R15,PR をある領域 (setjmp を呼び出したときの引数) に保存し 0 を返す。longjmp は逆にレジスタを復帰して 1 を返す。PR も保存/復帰するため、setjmp を呼び出した場所に戻る。

3.3 例外/割り込み処理モジュール

例外/割り込み処理モジュールは tlb_miss、general_exception,interrupt がある。それぞれ VBR + 0x400,0x100,0x600 に設定される。これらの処理はファイル entry.S に書かれており、exception_handling_table、interrupt_table で登録されている。テーブルは .section .data として書かれているので、RAM 上にとられる。

例外/割り込み発生に実行されるルーチンは entry.S にあるが、r0 〜 r7 のバンクレジスタの使用方法が異なる entry_non_scrach.S も用意してある。entry.S では、常にバンク 0 を使用するものと想定して gdb にバンク 0 のレジスタを送る。entry_non_scrach.S はバンク 1 も使用することを想定して、例外/割り込み発生時にアクティブなバンクのレジスタを gdb に送る。レジスタはバンク 0,1 共に破壊しない。その代り例外/割り込み発生した時、レジスタを退避するのにスタックを一時的に使用するため、スタックポインタの設定を誤るとリセットされる。entry.S ではスタック代りに バンク 1 の r0 と r1 レジスタを使用するため、スタックポインタの設定ミスが発生してもリセットされないが、バンク 1 のレジスタ r0 と r1 が破壊されるので、注意が必要である。

例外要因レジスタの値により RAM 上にある擬似ベクタテーブルに登録されているハンドラを実行する。”trapa #0x3f” で stub を呼び出すことによって疑似ベクタテーブルに例外ハンドラを自由に登録できる。なお疑似ベクタテーブルに登録されているプログラムを実行する前には、各レジスタを割り込み直後と同じ状態にしている。

割り込み要因が break として trapa もしくは、gdb からのシリアル経由のリモートブレークなら、割り込み前のレジスタを sh-stub.c で定義されている registers という配列に全て保存し、割り込みマスクを 15 に設定する。このとき、handle_exception() が実行され、BIOS コールかどうかを trapa の引数で判定する。BIOS コールなら handle_bios_call() が呼び出され、それ以外なら，gdb_handle_exception() が呼び出される。

アドレスロード/ストア例外で、その原因が stub によるメモリ操作なら longjmp を実行してメモリ操作直前に復帰する。

メインルーチンが終了すると、registers に格納されているレジスタを復帰して RTE によりユーザープログラムの実行を再開する。

3.4 メインルーチン (gdb_handle_exception 関数)

メインルーチンで使われるグローバル変数、関数は次の通りである。

グローバル変数

registers
stub が実行される前の各レジスタを保持する

dofault
メモリアクセスを行ったこと示すフラグ。setjmp を実行した後’0’ とする。この状態でアドレスロード/ストア例外が発生するとハンドラが longjmp を実行して setjmp 実行時の状態に戻す

remcomEnv
setjmp,longjmp でレジスタを保存/復帰するための長さ 9 の整数配列

stepped
’1’ でステップ処理を行っている事を示す

instrBuffer
ステップ処理により trapa に入れ替えたアドレスとデータを保存する

関数

void getpacket(char *buffer)
gdb からのコマンドをポーリングで待つ。コマンドを受け取るとチェックサムを計算して正しければ’+’ を返
コマンドを buffer に入れて返す。正しくなければ’-’ を返し再びコマンドを待つ。

void putpacket(register char *buffer)
buffer で渡されたコマンドとそのチェックサムを計算してパケットとして gdb に送る。gdb からの応答が’+
受け取るまでこれを繰り返す。

int computeSignal(int exceptionVector)
割り込み要因により、UNIX の SIGNAL に対応するシグナルナンバーを求めて返す。

void flush_icache_range(unsigned long start,unsigned long end)
ステップ処理でメモリ上の命令を書き換えた後、キャッシュをパージする。

void doSStep(void)
registers に保存されている PC すなわちユーザープログラムの次に実行する命令を読み、その命令に応じて
の次の実行番地のアドレスと命令を instrBuffer に保存した後、”trapa #0x20” を書き込む。

void undoSStep(void)
doSStep() によって trapa に書き込まれた番地へ instrBuffer に保存された命令によって元に戻す。

メインルーチンが開始される時、stub に制御が移動したことを示す”S:XXX” を gdb に送信し、無限ループの中
gdb からのコマンドをポーリングで待つ。この無限ループから抜けるのは、gdb からステップ、コンテーニューの
マンドが送られるか、メモリアクセス要求によるアドレスロード/ライト例外が発生した時である。

スタブが呼び出された原因が “trapa #0x20” ならステップ/gdb からのブレークポイントの設置により命令が置
換えられてるため、registers の PC の値を 2 減す。次に、undoSStep() を呼び出しステップ処理により入れ替えた
令を元に戻す。これ以降は while(1) ループにより gdb からのコマンドを待ちそのコマンドを実行する。各コマン
より実行する処理を次に示す。

?:要因問い合わせ
割り込み要因によりシグナルナンバーを計算して gdb に返す。

g:レジスタ参照
mem2hex() により registers に入れられているレジスタ値を hex に変換して gdb に返す。

G:レジスタ設定
hex2mem() により受信バッファのデータを registers にコピーし、OK を gdb に返す。

m:メモリ参照
setjmp() によりコンテキストを保存した後、dofault を”0” にして指定された番地から指定された長さ meme2
で出力バッファに入れ gdb に返す。setjmp() を行うのはアドレスロードエラーで例外が発生した場合復帰す
めでありこの場合は”E03” を gdb に返す。送られたパケット自体のフォーマットがおかしければ”E01” を返

M:メモリ書き込み
m の場合とほぼ等しい手順で hex2mem() により指定されたアドレスからデータを書き込む

c:コンテーニュー

コンテニューアドレスが指定されているなら、registers[PC] をそのアドレスに設定しリターンにより stub を抜ける。

s:ステップ

コンテニューアドレスが指定されているなら、registers[PC] をそのアドレスに設定する。次に、doSStep() により trapa により stub を抜けて実行する命令の次の命令を trapa に置き換え、リターンにより stub を抜ける。

4 BIOS CALL

上に述べた BIOS CALL では以下のようなものがある。

4.1 gdb のコンソールへの出力

r0 に 0x00, r4 に出力したい文字を入れ trapa #0x3f を実行するとその文字がホストの gdb のコンソールに出力される。

4.2 例外/割り込みの登録

SH3 の例外機構はベクタ方式ではなく、例外/割り込みが発生すると VBR +0x100, もしくは +0x600 番地から実行を開始する。そのため、stub は擬似的なベクタテーブルを持ち例外/割り込みが発生するとその要因を判定してこのテーブルに登録された番地より実行を開始する。このベクタテーブルは RAM 上に置かれているため、ユーザープログラムで受け取りたい例外/割り込みとそのハンドラを登録できる。登録は、r0 に 0x08, r4 に例外要因番号 (SH7709/A なら INTEVT2, SH7708 なら INTEV T に設定される番号), r5 にハンドラのアドレスを入れて trapa #0x3f を実行する。登録されたハンドラ実行時の状態は VBR +0x400, +0x600 に配置したプログラムが CPU 実行される状態と同じである。