GDB Agent for RTEMS

Till Straumann <strauman|at|slac.stanford.edu>

3/14/2005; last modified 7/29/2008

Contents

1 Introduction and Features

This document covers version 1.5 of the target agent.

1.1 Architecture

This gdb stub is an implementation of the remote end of the GDB *remote protocol* for a RTEMS target. Unlike other stubs available for

RTEMS (e.g., the m68k implementation), this version does NOT run at 'exception' level but is intended to be less intrusive. It's primary use is debugging of user-level / application code rather than the kernel itself (kernel debugging is still possible to a certain extent but the stub essentially needs a functional and initialized system to operate).

The stub runs as a 'daemon task' executing debugging commands as demanded by the host gdb. Unlike a 'low-level stub' which interrupts/stops the target system as a whole while talking to gdb, this implementation only affects the execution of one or multiple 'target threads' leaving other tasks alone.

As a consequence, it is possible to use TCP/IP as a communication protocol (no breakpoints in networking must be set, though!).

1.2   Multithreading

Gdb semantics expect the target system to be 'interrupted' or 'stopped' while listening to gdb commands. This implementation, OTOH, attempts to be as little intrusive as possible. For this purpose, a list of 'stopped' tasks is maintained internally. Tasks are interrupted only as a result of hitting a breakpoint, incurring an exception or an explicit request by the debugger. Interrupted tasks can be inspected as usual (stack frame, register contents etc.) and can be 'continued/resumed'. Only 'stopped' tasks can be inspected - the others continue executing normally.

The stub also creates a special helper task context which is always interrupted when the target is stopped, hence it is possible to do non-task related work (inspect/change memory, disassemble, ...) without having to stop an application task.

1.3   Cexp Support

A patch to 'gdb' is provided extending the remote protocol in order to support loadable modules. Gdb can inquire the list of currently loaded modules and their section addresses as well as load/unload additional modules on the fly.

1.4   Supported Targets

While many parts of the stub are intended to be portable and target-architecture independent, there are a few pieces of code which are BSP/CPU dependent. Currently, low-level support has been implemented for the i386/pc386, m68k/coldfire and powerpc/shared (and derived - should probably work on any 'new-exception processing' ppc BSP) BSPs.

Adding support for a new target should not be extremely hard. It mainly consists of writing two routines for packing/unpacking register values into an array formatted as expected by gdb. Also, a low-level exception handler needs to be written (mostly for mapping exception codes to signal numbers). E.g., the i386/pc386 specific file consists of 400 lines of quite straightforward code.

1.5 Supported Connection Methods

One of the strengths of the gdb remote protocol is its simplicity which makes it possible to use many character-stream based I/O methods for exchanging messages. Currently, the stub supports TCP/IP, serial port communication via RTEMS' termios driver and plain "dumb" character devices as long as they can be accessed via the file-system (`/dev/xzy`).

1.6 GDB Patches

One important design goal was not having to patch gdb itself. Unfortunately, it was not completely possible to meet this goal. However, the necessary patches are believed to be as small and as modular as possible.

- A small patch is needed to fix a bug [has been submitted as GDB bug #2029 but to-date, I don't know if it has been merged] in the powerpc stack unwinding.

- A small patch is needed to make it possible to obtain the list of tasks running on the target without interrupting and stopping them all (which would result in violating design goal #1: only interrupt tasks on request [explicit or breakpoint]).

- An additional patch adds a new gdb target ("`rtems-remote`") with protocol extentions to support loadable modules (CEXP). Note that the stub works fine without CEXP, however.

All of these patches are usually available with the source-code of the stub as a "`.diff`"-file of the format `gdb-<version>-<date>.diff`.

2 Building

For the prerequisites consult table 1

- Change directory to the gdb source topdir, create a build subdirectory (name of your choice), chdir there and configure (besides 'powerpc', the 'i386' and 'm68k' CPUs are supported and the cross-debugger may be configured accordingly)[1]:

---

[1]Older versions of the gdb patch (pre 6.8) required the target to be `<cpu>-ssrl-rtems`

host:	• download and unpack gdb-6.8 (or any later version for which a patch has been developed – check the `gdb-<version>-<date>.diff` patches distributed with rtems-gdb-stub
	• patches distributed with rtems-gdb-stub
target:	• (optional) libbspExt >= version 1.3
	• (optional) CEXP >= version 1.5.beta

Table 1: Prerequisites

```
>  mkdir build
>  cd build
>  ../configure --target=powerpc-rtems
```

Compile by typing `make` (gnumake) and install manually:

```
>  make
>  install -s gdb/gdb <target-dir>/powerpc-rtems-gdb
```

If you want to install everything including documentation you might need more elaborate `configure` options to get the install locations and the executable name right.

• The target agent is part of the `ssrlApps` package and should be build automatically as part of that package. If you don't want to use libbspExt (on powerpc) or Cexp you have to redefine the respective `USE_xxx` variables in the `Makefile`.

Note that if you *do* use those packages then you must make sure that your application executes the respective initialization routines (`bspExtInit()` and `cexpInit()`, respectively; consult the header files for any arguments and/or return value).

If you try to build *rtems-gdb-stub* independently from `ssrlApps`, an additional step is required because the agent needs a private header file, *cexpmodP.h*, from Cexp (only if Cexp support is to be used) that is *not* installed with the Cexp package: Edit the Makefile and point the `CEXP_SOURCE_PATH` variable to the correct location.

• As with any RTEMS application, you must not forget to link all required RTEMS managers – otherwise you incur what seem to be obscure errors. If you have plenty of memory then simply including all managers by setting `MANAGERS=all` in your `Makefile` is most comfortable.

3 Debugging Session

3.1 Prerequisites

Compilation

Code to be debugged must be compiled with the '`-g`' option. (preferrably everything)[2].

Optimization Caveat: It is possible to use GDB with optimized code (I do it all the time) but this may change the flow of execution, inline subroutine calls and may cause variables to 'disappear' etc.

Front-End

You want to use a GUI frontend together with cross-gdb, e.g.,

```
ddd --debugger <cross_arch>-rtems-gdb
```

Note: you need to set `PATH` prior to starting ddd/gdb, see below.

RTMF

GDB is a quite powerful program – make sure you know how to use it! The 'rtems-remote' target only adds 2 new commands. Everything else is off-the shelf GDB. Note that not all available commands necessarily make sense on this specific target.

Path Settings

Cross-debugging involves both, a host computer and the target system executing the actual code. The host computer needs access to *the same version* of the object files that the target computer is executing.

Upon connection to the target, the host computer obtains a list of currently loaded object files from the target. In order for the host GDB being able to locate the necessary object files, the `PATH` environment variable must point to the directories containing these objects. Note that GDB doesn't define a dedicated path variable for 'cross-architecture objects' but requires the user to set the ordinary `PATH` that is also searched for host executables! Note that `PATH` is an environment variable and cannot be changed from within a GDB session (GDB's `environ` and `path` commands only affect the `PATH` as seen by a native debuggee and don't have any effect in a cross-gdb session).

GDB also features a `load` command to instruct the target computer to load/link object modules. Since the target computer usually sees a

---

[2]Until recently, '`-g`' was implicitely set when building RTEMS proper. Unfortunately, this (late Nov. '05) no longer seems to be the case. I found that passing '`RTEMS_CFLAGS=-g`' on the command line to the `configure` script when building RTEMS did the job. I don't know if this is the recommended way, though. YMMV.

directory tree that is different from the host, any path information is discarded from the `load` argument – instead, the target's `PATH` environment variable must contain the directory where the target can locate the object that is to be loaded.

3.2   Starting the Debugging Agent

On the target, the debugging agent must be running. It needs to be linked into the system and is started by calling

```
int rtems_gdb_start(int agent_prio, char *dev_name);
```

Passing a priority of 0 lets the daemon pick its default scheduling priority. The second argument defines the connection method to be used. A NULL pointer lets the daemon listen on TCP port 2159 (registered port number for the gdb remote protocol) for an incoming connection, if a string is passed, it must be the path to a character device, e.g., `/dev/ttyS1`.

The agent can be stopped (`rtems_gdb_stop()`) and restarted with a different priority and/or communication method.

Note: rtems 4.6.2 requires a patch for this to work safely since the operation involves one task closing a socket on which another task is blocking.

3.3   Connecting to the Target

gdb can use either serial-port, TCP or UDP connections. Note that either uses certain resources on the target (e.g., the termios driver or the TCP/IP stack, respectively) and a debugging session may deadlock the system if the debugging agent needs a resource that is locked when the debuggee runs into a breakpoint.

The syntax for connecting to a target is (TCP)

```
(gdb) target rtems-remote <target>:2159
```

or (serial) – note that the agent's serial port runs at 115200 8N1 – use GDB's `remote baud` command:

```
(gdb) target rtems-remote <com_port_on_host>
```

UDP connections can also be used but gdb warns against them being less reliable. There is usually no point using UDP if TCP is available. One important exception would be if the target features a "standalone" UDP communication channel (e.g., a simple ethernet/IP/UDP packet driver/stack; such a driver/stack could be used to debug TCP/IP itself) or has reduced networking capabilities.

```
(gdb) target rtems-remote udp:<target>:2159
```

Note that `rtems-remote` is an extension of the standard GDB 're-mote' target – it adds support for Cexp modules (names, section addresses etc.).

Once connected to the target, the debugger is attached to a dummy thread context. All user and system threads are executing normally at this point.

Use the `detach` command to close the connection to the target at any time. All stopped threads will be resumed.

3.3.1 Sharing the Console Serial Line

If a priority less than zero is passed to `rtems_gdb_start()` the agent is executed in the context of the the caller, i.e., no separate task is created. Thus, the agent may be run from a shell (e.g., "Cexp") and take over the console line for the duration of the session, i.e., until the `detach` command is issued from GDB. In this "foreground" mode of operation, the agent terminates after a the session is ended returning control to the caller of `rtems_gdb_start()`.

A sample session example:

```
    Cexp>rtems_gdb_start(-1,0)
    GDB Daemon starting
< scrambled output since terminal mode changed >
    .y
< exit from terminal program and start gdb >
    (gdb) target rtems-remote < /dev/ttySxx >
    (gdb)    < session commands >
    (gdb) detach
    (gdb) quit
< restart terminal program and resume Cexp session >
    Cexp>
```

3.4 Switching Thread Context

In order to inspect a thread's stack, registers etc. it must be stopped. A thread is stopped either because it runs into a breakpoint or if the debugger is explicitly attached to it (`thread` command). If you want to attach to a specific thread you need to know its 'id' (i.e., GDB's id which is orthogonal to the RTEMS task id – use `info threads` to obtain a task list).

Once stopped, a thread remains suspended until you issue the `continue` command or `detach` (terminate the session). Either of those operations lets all stopped threads resume.

You can hit <Ctrl>-C to interrupt the target; this only interrupts the 'dummy'/'helper' thread and attaches GDB to it. I.e.,

`target rtems-remote xx:2159 //connect, attach to 'helper'`

At this point, the helper is stopped - you can inspect its registers etc. everyone else continues normally

```
info threads
t 14            //switch to thread ID 14
```

Thread 14 is stopped. The helper remains stopped. Inspect T14's registers + stack

```
t 12            //switch to thread ID 12
```

helper, T14 and T12 are stopped; inspect T12's registers + stack

```
t 14            //switch back to thread ID 14
```

helper, T14 and T12 remain stopped; inspect T14's registers + stack

```
cont            //resume everything
```

helper, T14 and T12 are resumed

```
<Ctrl>-C        //interrupt
```

interrupt, stop helper and attach to it - same state as after connection was established.

3.5   Executing Subroutines on the Target

When evaluating expressions at the (gdb) command line (e.g., within a call or print command) gdb may need to execute a subroutine on the target. For example,

```
(gdb) print printf("Hello\n")
```

invokes the printf routine on the target, retrieves the return value and prints it to the gdb console.

This is actually a pretty complex operation which involves creating a dummy stack frame and appropriate arguments etc. on the target. Some targets' exception handlers may not expect modifications of the stack and might crash as a result of an attempt to call a routine on the target.

Also, it should be noted that the routine is always executed in the context of the "current task". Hence, the "current task"

- must have sufficient stack space.

- must provide an appropriate execution environment. E.g., if the current task is an "integer-only" task then an attempt to invoke math routines may fail (crash).

- must not have been suspended due to an exception condition (i.e., it must not be a "dead" or "killed" thread).

The 'helper' task provides a suitable environment for the purpose of executing subroutines on the target.

3.6 DDD Note

After reloading any object on the target (using the `rtems load` or `rtems sync-objs` commands) the source code displayed by DDD may not be current (since the commands were performed at the GDB level, unnoticed by `ddd`). Use `ddd`'s `Reload Source` menu entry from the `Source` pull-down menu or the equivalent accelerator key combination `<Ctrl>-<Shift>-L` (default) to refresh the display.

4 RTEMS Specific GDB Commands

- use `rtems load` instead of `load`.

- use `target rtems-remote` instead of `target remote`. With statically linked applications the plain `remote` target is enough – the enhanced target version `rtems-remote` has added support for loadable modules: it queries the target for a list of loaded objects and their section addresses.

- `rtems sync-objs` (re)-synchronize gdb's file and section information with what's current on the target. Use this command if you added, removed or re-loaded modules from the Cexp> prompt instead of using `rtems load` from inside GDB.

- `info threads` semantics have slightly changed. Only a list of thread IDs/names is retrieved but not their stack frame or IP address (impossible without stopping all threads).

Note that even if you do not use Cexp or the `rtems-remote` *target you still must use a patched* `gdb` *in combination with this stub. One (small but important) change the patch achieves is avoiding to attach to all of the tasks running on the target (and thus stop them) which would bring the session to a grinding halt.*
    `<<TODO>>`
    More details; more commands; more implementation specifica
    `<</TODO>>`

5 Example Session

Assume you compiled a module in your home directory

```
cross-gcc -g -c blah.c -o /home/john/blah.o
```

You copy the object to a TFTP server where it can be 'seen' by the target.

```
cp blah.o /tftpboot
```

Prior to starting a debugging session, the PATH on the host is set up:

```
setenv PATH $PATH":/home/john"
```

On the target, (e.g., via startup script or from the Cexp> prompt) the PATH is also setup:

```
Cexp> setenv("PATH","/TFTP/BOOTP_HOST/:<more dirs>:",1)
```

Start a GDB session on the host

```
powerpc-rtems-gdb
```

Connect to the target

```
(gdb) target rtems-remote <target>:2159
// gdb obtains object file list and searches
        // PATH for 'blah.o'
```

Recompile 'blah.c'

```
cross-gcc -g -c blah.c -o /home/john/blah.o
cp blah.o /tftpboot
```

Reload module from GDB prompt

```
(gdb) rtems load blah.o
```