***gdb-spin* - GDB SpiNNaker Interface**

SpiNNaker Group, School of Computer Science, University of Manchester
Steve Temple - 8 Mar 2016 - Version 2.0.0

Introduction

This note describes ***gdb-spin***, which is an interface between GDB (the GNU Debugger) and SpiNNaker. *gdb-spin* is a simple Perl program which receives debugging commands from GDB and passes them to a specific core in a SpiNNaker system.

It is currently at a very primitive stage where the only thing it can do is allow inspection of the static variables of a program executing on SpiNNaker. Nonetheless, this is still a very useful thing to be able to do and facilitates debugging programs while they execute rather than having to stop them as is usual with GDB and similar debuggers.

You should note that *gdb-spin* is a software debugger and relies on the target system operating correctly if it is to be functional. If the target system is crashed, you will need to use a hardware (JTAG) debugger to do a port-mortem debug.

Principle of Operation

GDB defines a remote debugging protocol based on packets of ASCII text which are sent to and from a target system which is being debugged. Normally, these packets are sent directly to the target system via a serial line or using TCP/IP. The packets typically contain requests to read or write memory, set breakpoints or start and stop execution on the target system. In order for this to work, a GDB **debug stub**, which responds to the debug packets, is normally installed as part of the target application software.

Building a GDB debug stub is quite difficult to do in SpiNNaker and so *gdb-spin* uses a different strategy. GDB connects (via TCP/IP) to *gdb-spin* which runs on a host workstation. Debug packets from GDB are translated by *gdb-spin* which then forwards them to a specific core on the target SpiNNaker system as SCP (SpiNNaker Command Protocol) packets. In its present form, *gdb-spin* only forwards memory read requests which allows GDB to interrogate the memory on the selected core.

In order for GDB to know the position of variables in memory, it needs to have debug information generated during the SpiNNaker application build process. This will normally be contained in the ELF file generated in the build. It's important to note that the compilation (and assembly, if appropriate) steps need to be told to generate this information by giving them the `-g` (debug) flag when they are run. All libraries must also be (re)generated with this flag if their internal variables are to be visible.

Running *gdb-spin*

The essential steps in getting *gdb-spin* to work are to install it and start it running and then to start GDB and instruct GDB to connect to *gdb-spin*. It's also necessary to be able to tell *gdb-spin* which SpiNNaker core is being debugged and there is a simple interface, accessible from GDB, to do this.

gdb-spin uses the Perl libraries which come with the SpiNNaker low-level software tools to

connect to SpiNNaker. If you have these tools installed then you should ensure that *gdb-spin* is in the `.../tools/` sub-directory (and is executable). You should then be able to start it from the command line like this

```
unix> gdb-spin <spin_ip> [<port>]
```

You need to specify the IP address or hostname of the SpiNNaker system (`spin_ip`) and there is an optional TCP/IP port number which is the port on which it listens for requests from GDB. This defaults to 17899 and will probably only ever need to be changed if you want to run multiple instances of *gdb-spin* on the same host. If all is well when *gdb-spin* starts you should see several lines of information displayed on the terminal. There is currently no friendly way to terminate *gdb-spin* other than typing CTRL/C in its terminal.

gdb-spin commands

gdb-spin accepts commands from the user via the GDB command line. At present there are only two commands - `sp` which selects which SpiNNaker core to debug and `debug` which controls the level of debugging output from *gdb-spin* and is unlikely to be useful to most people. `sp` is similar to the same command in `ybug`, taking up to three numeric arguments to select a core on a SpiNNaker chip for debugging.

To issue a command from the GDB command line you need to prefix it with GDB's `monitor` command as below. This instructs GDB to pass the command directly to *gdb-spin*. The command `sp` without arguments will display the currently selected core. For example

```
(gdb) monitor sp
#  (1) Current core 0 0 0
(gdb) monitor sp 2 5 7
#  (1) Select core 2 5 7
```

Note that core (0, 0, 0) is selected by default when *gdb-spin* starts.

Multiple sessions

gdb-spin supports many (up to 25) simultaneous GDB connections so that you can be debugging many cores at the same time. Each new connection is allocated an identifier which is an integer which increments for each new connection. This identifier is displayed (in parentheses) in all output from *gdb-spin*.

Running GDB

Pretty much any (recent-ish) version of GDB will talk to *gdb-spin* and allow you to perform debugging. You may want to use the one which comes with the GNU software tools you are using to build SpiNNaker applications. This will (probably) be `arm-none-eabi-gdb`. Note that applications built with other tool chains (eg ARM compiler/assembler) can be also debugged provided that they generate ELF files in the usual format. The description below refers to command-line based debugging but there is no reason that GUI-based front ends to GDB would not work.

GDB can be started without arguments or you can specify the name of the ELF file that you wish to debug. You will see a prompt `(gdb)` and you can then issue debugging commands. The

first thing to do is to connect to the target system (via *gdb-spin*). This is done with the `target remote` command giving the host and port on which *gdb-spin* is running. In most cases, this will be the same machine as that on which GDB is running in which case the host can be omitted or specified as `localhost`. In the examples below, the first two are equivalent while the third debugs using an instance of *gdb-spin* on a remote machine.

```
unix> arm-none-eabi-gdb gdb-test.elf
GNU gdb (Sourcery CodeBench Lite 2013.05-23), etc, etc
(gdb) target remote :17899
(gdb) target remote localhost:17899
(gdb) target remote weasel.cs.man.ac.uk:17899
```

At this point, GDB can issue debug requests to *gdb-spin* which will then forward them to SpiNNaker. To leave GDB, you need to give the `quit` command and this will close the connection to *gdb-spin*.

A Simple Example

A simple example showing the use of GDB and *gdb-spin* will now be described. The program to be debugged is shown below. It is a simple API application which creates a 10ms timer whose callback just updates some static variables which can then be inspected with GDB.

```
// Simple SpiNNaker application to demonstrate use of GDB

#include <spin1_api.h>

// Declare a struct type to contain data for this program

typedef struct my_data
{
  uint count;
  uchar list[16];
  struct my_data *self;
} my_data_t;

// Declare an instance of the data struct. This is what we want to inspect
// with GDB. Placing many or all of the module's variables in a single
// struct is useful for debugging as we can display them all with a single
// command in GDB. The compiler is just as efficient at addressing these
// variables whether or not they are located in a struct.

my_data_t app_data;

// A timer function, called every 10ms. This copies the tick count into
// the data struct and updates one of the array elements with a new
// random number each time. It also sets the 'self' pointer to itself
// so that we can see this in GDB.

void timer_proc (uint ticks, uint arg2)
{
  app_data.count = ticks;
  app_data.list[ticks % 16] = spin1_rand ();
  app_data.self = &app_data;
}
```

```
// Main function which sets up and runs the timer

void c_main (void)
{
  spin1_callback_on (TIMER_TICK, timer_proc, 1);
  spin1_set_timer_tick (10 * 1000);
  spin1_start (SYNC_NOWAIT);
}
```

The program source is `gdb-test.c` and it is compiled to an APLX file and loaded onto a SpiNNaker core with `ybug` using the following commands. Note the use of the `CFLAGS` parameter in the `make` command which causes debug information to be placed in the ELF file which is also created by the make process.

```
unix> make APP=gdb-test CFLAGS=-g
unix> ybug 192.168.240.38
192.168.240.38:0,0,0 > app_load gdb-test.aplx . 1 16
192.168.240.38:0,0,0 > ps
Core State  Application       ID   Running  Started
---- -----  -----------       --   -------  -------
  0  RUN    scamp-134          0   0:12:07   6 Jan 12:05
  1  RUN    gdb-test          16   0:00:03   6 Jan 12:17
  2  IDLE   sark               0   0:12:07   6 Jan 12:05
  3  IDLE   sark               0   0:12:07   6 Jan 12:05
```

At this point the program is loaded and running on core 1 of chip (0, 0) of a SpiNNaker system whose IP address is `192.168.240.38`. We can now use GDB to inspect its state. First we start *gdb-spin* telling it to talk to `192.168.240.38`. This will need to be done in a new shell (as will the subsequent GDB session).

```
unix> gdb-spin 192.168.240.38
#-------------------------------------------------------------------------------
#
# SpiNNaker GDB Interface - Version 0.01 (experimental!)
#
# Connecting to SpiNNaker   192.168.240.38
# Starting GDB interface    0.0.0.0:17899
#
#-------------------------------------------------------------------------------
#
```

Next we start GDB, instruct it to connect to *gdb-spin* and load the relevant ELF file. We also give a command to make the printing of structs a bit nicer. Finally, we issue a command to *gdb-spin* to select the core on which our program is running.

```
unix> arm-none-eabi-gdb
GNU gdb (Sourcery CodeBench Lite 2013.05-23), etc, etc
(gdb) target remote :17899
(gdb) symbol-file gdb-test.elf
Reading symbols from /tmp/spinnaker_tools_134/apps/gdb/gdb-test.elf
(gdb) set print pretty on
(gdb) monitor sp 0 0 1
#  (1) Select core 0 0 1
```

GDB is now all set up to interact with the target core. To inspect a variable you use the `print` command. This will cause GDB to read the variable from SpiNNaker memory and print it on

the terminal. The example below reads the `app_data` variable three times over a period of a few seconds so that the change in the variable's value is obvious. Adding the `/x` qualifier to `print` causes output in hexadecimal. The fourth command reads a single element of the struct.

```
(gdb) print app_data
$1 =
  count = 1836203,
  list = "\b\350\255*\035:\212#r\262\217\033\301",
  self = 0x400360 <app_data>

(gdb) print/x app_data
$2 =
  count = 0x1c06d0,
  list = 0x3b, 0x73, 0xfe, 0xb5, 0x28, 0x97, 0xb4, 0x4c, 0x50, 0x8b, 0xfe,
    0xe3, 0x36, 0x6e, 0x62, 0xc3,
  self = 0x400360

(gdb) print/x app_data
$3 =
  count = 0x1c08b0,
  list = 0xac, 0x4f, 0x7e, 0x30, 0xa3, 0x38, 0x8a, 0x57, 0x23, 0xf, 0x1b,
    0xf6, 0x63, 0x81, 0xd6, 0x5a,
  self = 0x400360

(gdb) print app_data.self
$4 = (struct my_data *) 0x400360 <app_data>
```

You should note that if you change the program and recompile and reload it, it's possible that the ELF file you have loaded in GDB will become inconsistent with the newly loaded code and so it's wise to reload the ELF (using the `symbol-file` command) each time you do this.

The GDB command line interpreter performs command and variable name auto-completion and maintains a command history so is quite easy to use.

Current Limitations

The main limitation of the current implementation of *gdb-spin* is that it can only read and display static variables. In order to safely read variables on the stack the program must be stopped (so that the stack pointer is not moving around). This is almost certain to be fatal for most SpiNNaker programs which have real-time constraints. If there is no requirement for the program to continue after it has been stopped then it would probably be possible to provide this facility.

Similarly, providing a breakpoint mechanism would be possible but this would also stop program execution and so would only be useful in a limited range of circumstances. I would be happy to look at this if it was thought to be useful.

Finally, because GDB/*gdb-spin* needs to read SpiNNaker memory and needs the application processor to be up and running to do this, none of this will work if the application core has crashed badly and is unable to respond to SCP requests from the monitor processor.

Change log:

- *0.01 - 05jan15 - ST* - preliminary draft - comments to *steven.temple@manchester.ac.uk*

- *2.0.0 - 08mar16 - ST* - first release