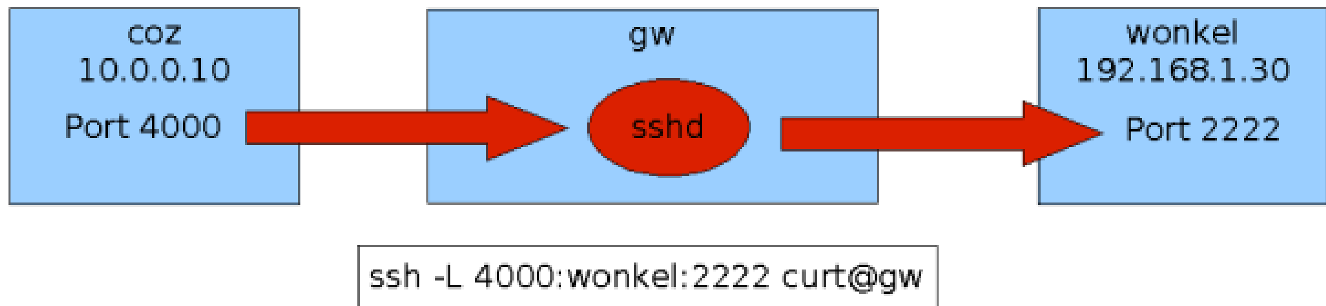
Talking Dirty with GDB and SSH Tunneling

Ever debugged a program remotely and felt like telling your computer where to go and how to get there? Hopelessly adding calls to printf() and recompiling as a steady string of explectatives flow from your over-caffeinated brain waves.

Fear not! Help is on the way. Read on to learn how to use gdbserver and ssh tunnelling to debug remote processes.



In embedded systems development making do with less is the name of the game – less CPU power, less physical RAM and less peristant storage (if any!) to name a few. Debugging a misbehaving process in this environment can be challenging, but a little ingenuity coupled with plenty of free software eases the problem.

In this article I will explain how to use GDB for debugging remote processes running on embedded systems from our desktop workstation.

For the purposes of this article the embedded system is a RPX_Lite from EmbeddedPlanet, which I discussed in a previous article. This board has a blindingly fast (not) 80MHz 823e PowerPC processor with 16MB of RAM. It's pretty cute, a 3 inch square that includes ethernet, USB, serial and a PCMCIA slot. Just right for experimenting.

Major Differences

While many differences exist between desktops and embedded systems running GNU/Linux, the biggest difference is size and power. Embedded systems have very specific design goals limiting the overall power consumption and physical dimensions. This leads to the use of low power CPUs, limited physical RAM and little or no persistant storage devices.

The next major difference is the CPU architecture – embedded systems often use low power CPUs that are not x86 based. To compile programs from your x86 based desktop you need "cross-compilation tools", which run on your local desktop, but generate excutable code for the target architecture. In this article I will be cross-compiling for the PowerPC architecture.

The last major difference is perspective. You will be debugging a process that is executing on a remote CPU not your local workstation. This requires a slightly different mindset then traditional debugging.

ELF and Binutil Background

Before getting to the nuts and bolts here's a quick review about how executable code and debugging information is stored in an ELF binary. Most modern *NIX systems use the the ELF format for executables and shared libraries.

On a GNU/Linux system a family of utilities called binutils exists for examining and manipulating ELF objects. In a cross-compiling development environment the usual tool names like gcc, gdb and all the binutils will have a prefix that describes the target architecture.

In this article I'm targeting the PowerPC 823e and the tool prefix is "ppc_8xx-". I am using the wonderful Embedded Linux Development Kit (ELDK), which has complete toolchains for the PowerPC. Let's play around with some of the binutils using a simple "Hello World" application:

```
#include

int main(void) {

  printf("Hello World\n");

  return 0;

}
```

First let's compile the program with debugging symbols using the -g option.

```
ppc_8xx-gcc -g -o hello hello.c
```

Note I used the cross compiler, ppc_8xx-gcc, to compile the program for the PowerPC 823e target. On my system the resulting binary size is 20632 bytes.

To see what symbols the binary contains use the nm binary utility. Remember I'm using the PowerPC version of nm, ppc_8xx-nm.

```
coz:~/articles/rgdb$ ppc_8xx-nm hello
   10010868 D _DYNAMIC
   10010948 T _GLOBAL_OFFSET_TABLE_
   [... stuff deleted]
   1001085c W data_start
   10000408 t frame_dummy
   1000048c T main
   100109d4 b object.2
   10010860 d p.0
            U printf@@GLIBC_2.0
```

The interesting lines for us are near the bottom:

```
1000048c T main
         U printf@@GLIBC_2.0
```

The first line shows the address of the main() function is 1000048c. "T" means the text section, which is an old term for the section where the code resides. The next line shows that printf() is an unresolved symbol and will be loaded from a shared library at run time. Interesting.

The ELF format defines sections where various information about the executable is stored. The most interesting sections are:

```
text -- where the executable code lives
```

data -- where global variables live
rodata -- where global "read only" constants live

In addition several other sections also are present, including sections containing the debugging information. To see all the sections and their sizes use the objdump binutil with the -h option to display section headers.

coz:~/articles/rgdb$ ppc_8xx-objdump -h hello

hello: file format elf32-powerpc

Sections:

```
Idx  Name          Size      VMA       LMA       File  off  Algn

0 .interp       0000000d  10000114  10000114  00000114  2**0 CONTENTS, ALLOC, LOAD, READONLY, DATA
1 .note.ABI-tag 00000020  10000124  10000124  00000124  2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA
2 .hash         00000030  10000144  10000144  00000144  2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA
3 .dynsym       00000070  10000174  10000174  00000174  2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA
4 .dynstr       0000007a  100001e4  100001e4  000001e4  2**0 CONTENTS, ALLOC, LOAD, READONLY, DATA
5 .gnu.version  0000000e  1000025e  1000025e  0000025e  2**1 CONTENTS, ALLOC, LOAD, READONLY, DATA
6 .gnu.version_r 00000020  1000026c  1000026c  0000026c  2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA
7 .rela.dyn 0000000c  1000028c  1000028c  0000028c  2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA
8 .rela.plt 00000030  10000298  10000298  00000298  2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA
9 .init 00000028  100002c8  100002c8  000002c8  2**2 CONTENTS, ALLOC, LOAD, READONLY, CODE
10 .text 00000528  100002f0  100002f0  000002f0  2**2 CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .fini 00000020  10000818  10000818  00000818  2**2 CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .rodata   00000024  10000838  10000838  00000838  2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA
13 .sdata2   00000000  1000085c  1000085c  0000085c  2**2 CONTENTS, ALLOC, LOAD, READONLY, DATA
14 .data 00000008  1001085c  1001085c  0000085c  2**2 CONTENTS, ALLOC, LOAD, DATA
15 .eh_frame 00000004  10010864  10010864  00000864  2**2 CONTENTS, ALLOC, LOAD, DATA
16 .dynamic  000000c8  10010868  10010868  00000868  2**2 CONTENTS, ALLOC, LOAD, DATA
17 .ctors00000008  10010930  10010930  00000930  2**2 CONTENTS, ALLOC, LOAD, DATA
18 .dtors00000008  10010938  10010938  00000938  2**2 CONTENTS, ALLOC, LOAD, DATA
19 .jcr  00000004  10010940  10010940  00000940  2**2 CONTENTS, ALLOC, LOAD, DATA
20 .got  00000014  10010944  10010944  00000944  2**2 CONTENTS, ALLOC, LOAD, CODE
21 .sdata00000000  10010958  10010958  00000958  2**2 CONTENTS, ALLOC, LOAD, DATA
22 .sbss 00000000  10010958  10010958  00000958  2**0 ALLOC
23 .plt  00000078  10010958  10010958  00000958  2**2 ALLOC, CODE
24 .bss  0000001c  100109d0  100109d0  00000958  2**2 ALLOC
25 .comment  0000016e  00000000  00000000  00000958  2**0 CONTENTS, READONLY
26 .debug_aranges 00000020  00000000  00000000  00000ac6  2**0 CONTENTS, READONLY, DEBUGGING
27 .debug_pubnames 00000040  00000000  00000000  00000ae6  2**0 CONTENTS, READONLY, DEBUGGING
28 .debug_info    00001f81  00000000  00000000  00000b26  2**0 CONTENTS, READONLY, DEBUGGING
29 .debug_abbrev 00000271  00000000  00000000  00002aa7  2**0 CONTENTS, READONLY, DEBUGGING
30 .debug_line    00000232  00000000  00000000  00002d18  2**0 CONTENTS, READONLY, DEBUGGING
31 .debug_frame   0000003c  00000000  00000000  00002f4c  2**2 CONTENTS, READONLY, DEBUGGING
32 .debug_str00000041  00000000  00000000  00002f88  2**0 CONTENTS, READONLY, DEBUGGING
```

WOW! That's a lot of sections and many of them contain debug information. These sections can add quite a bit of size to an executable and none of it is essential to running the program. The information is only useful when trying to debug.

Aside: the ctors and dtors sections are for "constructors" and "destructors", like those used for static C++ objects.

In order to save as much space as possible on an embedded system we often strip off all the non-essential information from the ELF sections. The binutil strip does just this.

```
coz:~/articles/rgdb$ ppc_8xx-strip hello
```

Now the size of my executable is 4092 bytes, a reduction of 16540 bytes. That is over an 80% reduction – awesome! But it comes at a price. Without the debug sections debugging will be impossible... Sort of. More on that later.

Remote Debugging With GDB

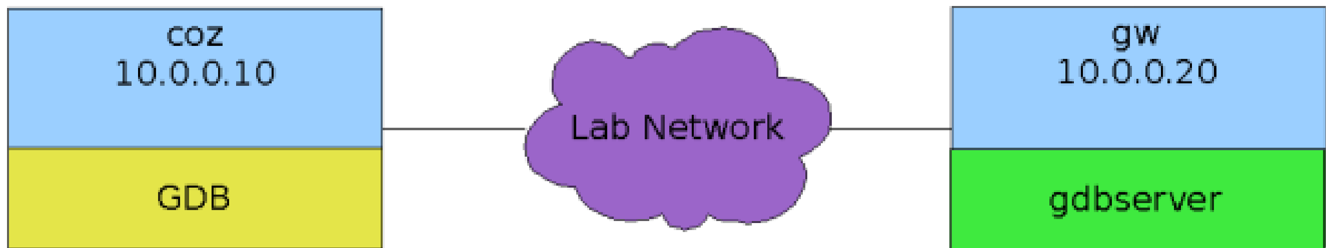
So now we have a stripped application that we can execute on our embedded system. But what if it is crashing and we want to debug it? A couple of obsticles sit in our way.

First, the target system has limited storage so we did not bother to put a cross-compiled version of GDB on it. On my development workstation the GDB executable is 9973975 bytes, nearly 10 megabytes. Clearly that won't leave much room for anything else if I only have 16MB total on the embedded system.

What to do?

The answer is to use gdbserver, a small footprint server that implements the low level features of GDB.

Consider the following diagram – using TCP the feature-rich GDB on my workstation connects to the light-weight gdbserver running on the embedded system. Most of the heavy lifting is done by the GDB on my workstation, while gdbserver deals with the low level interactions.



On my system gdbserver is only 59303 bytes, a considerable improvement over the size of the full GDB program.

In the following examples my workstation, coz, has the IP address of 10.0.0.10 and the embedded system, gw, has an IP address of 10.0.0.20 as shown in the above diagram.

The first step is to attach the gdbserver to a process on the target system. You can have gdbserver start a program and attach to it immediately or you can attach to an already running process using the process ID (pid). The last argument you need to specify is the TCP port that gdbserver will listen on. Here's the syntax:

```
gdbserver host:2222 PROGRAM [ARGS...]
gdbserver host:2222 --attach PID
```

In the above examples the gdbserver would listen on port 2222. Using gdbserver to launch our hello world application on the embedded system looks like this:

```
gdbserver host:2222 hello
Process hello created; pid = 23125
Listening on port 2222
```

The prompt does not comeback as the gdbserver is now blocking, waiting for connections on port 2222.

The next step is to start the main GDB program on your workstation and connect to the gdbserver process. In order for the main GDB debug my program it needs to examine the "unstripped" version of executable that contains all of the debugging symbols. The simplest thing to is to chdir to the directory containing the unstripped executable and start the cross compiled version of the gdb, like this:

```
coz:~$ cd ~/articles/rgdb
coz:~/articles/rgdb$ ppc_8xx-gdb

GNU gdb Yellow Dog Linux (5.2.1-4b_4)
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions.

Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i386-redhat-linux --target=ppc-linux".

(gdb)
```

To "tell" GDB to read the symbols from the unstripped executable use the GDB file command like this:

```
(gdb) file hello
Reading symbols from /home/curt/articles/rgdb/hello...done.
```

If your application uses shared libraries and most real world applications do, then you also need to tell GDB where to locate these libraries. These need to be the unstripped versions of these libraries so that GDB can tell you more info.

Set the GDB "solib-search-search-path" variable so that GDB can find the shared libraries used by your application, like this:

```
(gdb) set solib-search-path  [path to libraries]
```

If your application has a lot of shared libraries spread all over your source tree (and most real world ones do) then here's a little trick for the solib-search-path variable. Create one directory and populate it with symlinks to all of your shared libraries. Then you need only specify this one directory when setting the solib-search-path variable. Comes in handy.

Now we are ready to connect to the gdbserver running on the embedded system. We use the target remote command from the main GDB command prompt, like this:

```
(gdb) target remote 10.0.0.20:2222
Remote debugging using 10.0.0.20:2222
0x10000120 in ?? ()
```

On the embedded system console you should see this output:

```
Remote debugging from host 10.0.0.10
```

Now we are connected and the program being debugged is currently paused. Now would be a good time to set some break points and then continue running the program. Here's an example:

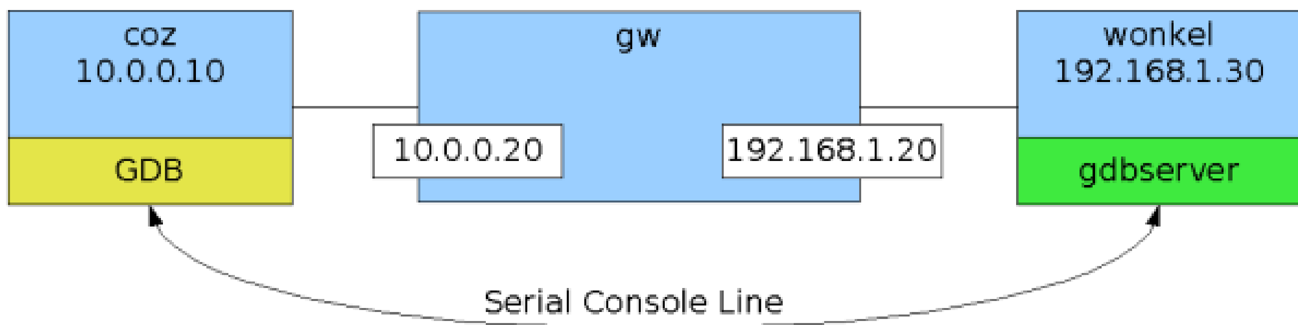
```
gdb) b main
Breakpoint 1 at 0x1000048c: file hello.c, line 4.
gdb) continue
Continuing.

Breakpoint 1, main () at hello.c:4
 4         printf("Hello World\n");
```

And there we are! We are remotely debugging the stripped executable. Pretty cool, huh? I love it! You can now use all your favourite GDB commands and techniques to debug.

Personally I like running GDB from within Emacs, but your tastes may vary. You can use any GDB front-end you want for remote debugging. Very nice.

SSH Tunneling and GDB

Suppose you have the network topology shown in the following diagram and you want to debug a process running on the host wonkel. In this topology the host gw is dual homed with one interface on the 10.0.0.0/24 network and one on the 192.168.1.0/24 network. The host wonkel is also on the 192.168.1.0 network, while your workstation is on the 10.0.0.0 network.



The diagram also depicts a serial console connection from coz to wonkel – serial UART connections are very common on embedded systems for debug purposes, a back door for when the network connection is not working. Even though they are a bit slow, UARTs are cheap, reliable and easy to configure. A serial console is usually the first bit of hardware tested out when bringing up a new board. The problem here is that no routable network path exists from coz to wonkel – gw cannot act as a gateway and forward packets in the normal manner.

What to do? I'll be honest – a lot solutions present themselves. You could configure iptables on gw to forward packets from coz to wonkel. That works fine if you have root access to gw.

Another method is to use the port forwarding capabilities of your old friend, ssh. The trick here is to forward connections on a local coz port to a port on wonkel – as a side benefit the traffic on the tunneled port is also encrypted by the SSH protocol.

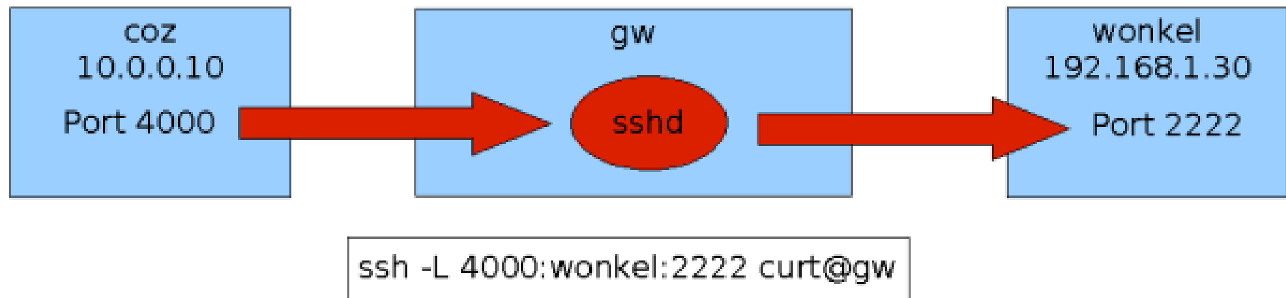
Via the serial console you can login to wonkel and perform basic commands. Using the serial console start the gdbserver on wonkel, just like the previous example:

```
gdbserver host:2222 hello
Process hello created; pid = 23125
Listening on port 2222
```

Now we need to create an SSH tunnel from coz to wonkel via gw. Here's the command to do that:

```
ssh -L 4000:wonkel:2222 curt@gw
```

This opens a listening TCP socket on the local host, coz:4000. Whenever a connection is made to this socket it is forwarded to the sshd process on gw, which then opens a connection to wonkel:2222. This is shown in the following diagram.



Now we can start GDB on coz like before, but this time the "target host" command use localhost:4000, like this:

```
(gdb) target remote localhost:4000
Remote debugging using localhost:4000
0x10000120 in ?? ()
```

This connection is tunneled via gw to wonkel:2222. On the wonkel serial console you should see:

Remote debugging from host 192.168.1.20

Note wonkel thinks the debug request is coming from gw, not coz.

Now you can proceed to debug as before.

I hope you have found this intro to remote debugging and ssh tunnelling useful. All comments are welcome. Happy hacking!