Debugging Using gdbserver

Sometimes a program has no technical errors that cause the compile to fail but fails to meet the developer's expectations when run. This is typically due to algorithm or data structure design errors which can be difficult to find with just visual inspection of the code. Because of this, it can be beneficial to run a debugger targeting the binary resulting from the compile process. Debugging is the process of watching what is going on inside of another program while it is running. When a program is compiled with debug symbols included in the binary, it is possible to observe the source code and corresponding assembly while running the debugger.

When working with embedded systems the binary is usually compiled on a development machine with a different CPU architecture than what is on the target machine. This can be a problem when, as is typically the case, the target machine lacks the system resources to run a debugger. In these cases, it is possible to use the GNU debugger, or GDB, on the development machine to remotely debug the target machine provided it has a program called `gdbserver`. All EMAC OE builds are packaged with `gdbserver` to simplify the setup process for developers.

This guide is intended to build a basic understanding of how to use `gdbserver` with EMAC products. It is not intended as a general guide to debugging computer programs. For help with that, see the GDB man pages on the development system or read this manual (http://sourceware.org/gdb/current/onlinedocs/gdb.html) on debugging with GDB.

Table 1. Conventions Used

`target_program`	The name of the application being debugged. This is the result of Makefile build process.
`target_machine`	Connection information for the target machine. This can either be a serial port (ie. `/dev/ttyS2`) or a TCP connection in the form of HOST:PORT.
`/path/to/sdk/`	Represents the development system path to the EMAC OE SDK.

Setup

Using `gdbserver` involves setting up both the target machine and the development machine. This requires that the binary application be present on both development and target machines. The development machine copy of the application must be compiled with debug flags whereas this is not strictly necessary for the target machine. See the Optional ''global.properties'' Modifications Section on the New EMAC OE SDK Project Guide for more information. See the EMAC OE Getting Started Guide for more information on how to connect to the target EMAC product using a serial port or Ethernet connection.

Target Machine

Because EMAC OE builds are distributed with `gdbserver`, installation is not a concern. The only setup necessary is to run `gdbserver` with `target_program`:

1. If the target application is already running, use the `attachpid` option to connect `gdbserver` to the application as shown below. The `PID` argument can be determined using `pidof`.

```
$ gdbserver target_machine --attach PID
$ pidof PID
```

2. If the target application is not already running, the name of the binary may be included as an argument to the `gdbserver` program call.

```
$ gdbserver target_machine target_program [ARGS]
```

This establishes a `gdbserver` port on the target machine that listens for incoming connections from GDB on the development machine. In debug terminology, `gdbserver` is "attached" to the process ID of the program being debugged. The next step is to run GDB on the development machine using the `target_program`.

Development Machine

1. First, `cd` to the directory where the target executable is stored.
2. Run the EMAC OE SDK GDB:

```
$ /path/to/sdk/EMAC-OE-arm-linux-gnueabi-SDK_4.0/gcc-4.2.4-arm-linux-gnueabi/bin/arm-linux-gnue
```

3. Run the following commands in GDB to prepare for the debug session:

```
(gdb) target remote target_machine
```

Note that the location of the GDB in the toolchain may differ from what is shown above depending on which version of the SDK is used.

Sample GDB Session

This example GDB session uses the EMAC OE SDK example project named `pthread_demo`. It consists of the single source file `pthread_demo.c`. The program is called with a single integer argument indicating how many `reader` threads the user wishes to create. The following describes the tasks of the `main` thread:

1. The `main` thread performs user input validation. It prints a usage message according to
2. The `main` thread initiates a new thread that uses the `generator()` function to perform the following tasks:
    1. Checks to see if the number of `reader` threads matches the number of times a `reader` thread has acquired the mutex `lock` and performed its task. If the two values do match, then the `generator` thread unlocks the mutex, breaks out of the while loop and moves on to line 167 to gracefully exit. If the two values do not match, then the `generator` thread continues through the rest of the while loop described in steps 2.2 and 2.3.
    2. Generates random data to be stored in the `data` struct shared by all the threads. To do this, it protects the `data` struct with the use of a mutex variable.
    3. Sleeps after giving up its lock on the mutex so that another thread might have a change to acquire the lock.
3. After creating the `generator` thread the `main` thread iteratively as many `reader` threads as indicated by the single integer argument. Each `reader` thread performs the following tasks:
    1. Waits for a chance to acquire the mutex `lock`. Once the mutex `lock` is acquired, it prints the value of the random number generated by the `generator` thread in its last run.
    2. Increments an integer in the `data` struct to indicate that it has completed its task.
    3. Gives up its lock on the mutex and exits.
4. After creating the prescribed number of `reader` threads, the `main` thread then waits for each thread created to exit gracefully.
5. The `main` thread exits.

The SDK version of `pthread_demo.c` works according to the description above with a `MAX_THREAD` value of 100. However, for the purpose of this example debug session it is instructive to use a faulty version of the same program. Replace lines 75-80 in `pthread_demo.c` with the code snippet shown in Listing 1 below.

Listing 1. `pthread_demo.c` Modification

```
	if ((data.num_threads < 1) || (data.num_threads > MAX_THREAD)) {
		fprintf(stderr,
			"The number of thread should between 1 and %d\n",
			MAX_THREAD);
		exit(EXIT_FAILURE);
	}
```

Useful GDB Commands

The following is a brief description of some essential GDB commands. Each description is followed by a link to the official GDB documentation page that has more specific information about what the command does and how to use it. Please note that the official GDB documentation is targeted for the latest GDB release which at the time of writing this documentation is 7.2. The version of GDB that EMAC distributes with the OE products, however, is version 6.8. Because of this, the links to documentation below may provide slightly different information. The biggest difference between the two version of GDB, however, is in the support for debugging programs with multiple threads. This is reflect in the documentation as well. Because of this, EMAC has set up ftp access to GDB 6.8 documentation on its web server. It is highly recommended that the GDB 6.8 documentation be referenced in cases where the program does not seem to support commands or options specified in the current official documentation.

- **`start`/`run`**
  These commands are used to start the debugged program with the only difference being that `start` automatically pauses execution at the beginning of the program's `main` function whereas `run` must be told explicitly where to pause using the `breakpoint` command listed below.
  See Debugging with GDB, Section 4.2: Starting your Program (http://sourceware.org/gdb/current/onlinedocs/gdb.html#Starting)
- **`kill`**
  Used to kill the currently-running instance of `target_program`.
  See Debugging with GDB, Section 4.9: Killing the Child Process (http://sourceware.org/gdb/current/onlinedocs/gdb.html#Kill-Process)
- **`print`**
  Used to print the value of an expression.
  See Debugging with GDB, Section 10: Examining Data (http://sourceware.org/gdb/current/onlinedocs/gdb.html#Data)
- **`list`**
  List contents of function or specified line.
  See Debugging with GDB, Section 9: Examining Source Files (http://sourceware.org/gdb/current/onlinedocs/gdb.html#Source)
- **`layout`**
  This is a TUI (Text User Interface) command that enables the programmer to view multiple debug views at once including source code, assembly, and registers.
  See Debugging with GDB, Section 25.4: TUI Commands (http://sourceware.org/gdb/current/onlinedocs/gdb.html#TUI-Commands)
- **`disassemble`**
  This command allows the programmer to see assembler instructions.
  See Debugging with GDB, Section 9.6: Source and Machine Code (http://sourceware.org/gdb/current/onlinedocs/gdb.html#Machine-Code)
- **`break`** This command specified a function name, line number, or instruction at which GDB is to pause execution.
  See Debugging with GDB, Section 5.1: Breakpoints (http://sourceware.org/gdb/current/onlinedocs/gdb.html#Breakpoints)
- **`next`/`nexti`**, **`step`/`stepi`**
  Allow the programmer to step through a program without specifying breakpoints. The `next`/`nexti` commands do not step into functions but instead run through function calls entirely, stopping on the next line the the same stack frame; `step`/`stepi`, on the other hand, do step into function calls, stopping on the first line in the next stack frame. The difference between `step`/`next` and `stepi`/`nexti` is that the `i` indicates

instruction-by-instruction stepping.
See Debugging with GDB, Section 5.2: Continuing and Stepping (http://sourceware.org/gdb/current/onlinedocs/gdb.html#Continuing-and-Stepping)

- **`continue`**
  Used to continue program execution from the address where it was last stopped.
  See the Debugging with GDB link for `next`/`step` for more information about the `continue` command.
- **`bt`**
  Short for `backtrace`, which displays to the programmer a brief summary of execution up to the current point in the program. This is useful because it shows a nested list of stack frames starting with the current one.
  See Debugging with GDB, Section 8.2: Backtrace (http://sourceware.org/gdb/current/onlinedocs/gdb.html#Backtrace)

Session Walk-through

This debug session walk-through assumes that the program has been compiled using the modified source code above and that both the target machine and the development machine have been set up according to the above Setup section. The walk-through is divided into multiple "lessons" with the intent of first introducing the use of the commands described above and then actually running GDB to debug a known programming problem. Each lesson may be run independently of the others, but it is recommended that each be run in order starting from Lesson 1 for the first time through.

Lesson 1: Navigation and Code Display

This lesson assumes that `gbserver` has been run as in the Target Machine Setup section above with an `ARG` value of 3. Other values are fine so long as they fall within the range of 1 to 100. The number '3' was arbitrarily chosen to avoid having to use a symbolic variable in the explanations below.

1. Type `b main` to set a breakpoint at the `main` function in the source code.
2. Type `continue`. This will cause the program to continue from the breakpoint set by GDB at startup. The program was passed an argument of 3, indicating that three threads should be created.
3. Type `b 73` to set a breakpoint at line 73 in the source code, which should be the line containing `data.num_threads = atoi(argv[1]);`. The letter 'b' is an alias for `break`.
4. Type `continue`. The program will continue execution up until line 73 in the source code. At this point, type `layout split` to view a split screen containing both the source code and the assembly-level machine instructions. Both screens show the program's current location in execution. The assembly-level display shows what the target's processor is actually executing at that point in the source code as shown in the source-level display. To view either of these without the other type `layout asm` for just assembly-level and `layout src` for just source-level.
5. Type `nexti`. This will cause the program to execute the next instruction in the current stack frame which is a `mov` instruction beginning to prepare the current stack for a call to the library function `atoi()`. The details of this process are beyond the scope of this tutorial; essentially, the program needs to store information about the current execution location in the stack for when the `atoi()` function finishes. Type `ni` (alias for `nexti`) three more times. You should end up on a `bl` instruction in the assembly view as shown in Listing 2 below. The source layout should still show the program on line 73.
   Listing 2. GDB Assembly Layout

```
B+ |0x887c <main+112>       ldr    r3, [r11, #-84]                  |
   |0x8880 <main+116>       add    r3, r3, #4      ; 0x4            |
   |0x8884 <main+120>       ldr    r3, [r3]                         |
   |0x8888 <main+124>       mov    r0, r3                           |
  >|0x888c <main+128>       bl     0x86e0 <atoi>                    |
```

Note that the assembly may look different depending on the target architecture.

6. Type `stepi`. This will cause the program to move into the next stack frame and GDB to show the assembly-level instructions of the `atoi()` call. Since the library containing `atoi()` was likely not compiled with debug symbols, the source-level layout will show the message `[ No Source Available ]`.
   Note that if you had instead typed `nexti` the program would have executed all the relevant instructions in `atoi()` and paused only on the next instruction in the current stack frame.
7. Type `bt`. This will cause the program to display a human-readable version of the current stack. Each stack "frame" is represented by the name of the function call it represents with that function's location in memory. Type `bt full` to get a list of the variables local to each stack frame.
8. Type `finish`. This will cause the current stack frame to return and execution to pause on the next instruction of the previous stack frame.
9. Type `kill`. This will cause the current process to be killed by `gdbserver` at the target machine. `gdbserver` will also terminate at this point. In order to start a new remote debug session, start `gdbserver` as described in the Target Machine Setup section and re-run step 3 of the Development Machine Setup section.

Lesson 2: Finding the Bug

Though this sample is contrived, it is still useful to demonstrate how to find a design mistake in an otherwise well-written (no errors or warnings) program. These types of mistakes typically have to do with the array boundary miscalculations, logic and comparison operator mistakes, or other simple mistakes. For the sake of demonstration, assume that the actual mistake is unknown. This lesson assumes that `gdbserver` has just been started as in the Target Machine Setup above with an `ARG` value of 5.

1. Before starting the program in the debugger again, run it by itself on the target machine to see what the actual program output is:

```
$ /tmp/pthread_demo 5
The number of thread should between 1 and 100
$
```

   The program was given an input of '5' yet the output message seems to indicate that this is out of range which is obviously not true.

2. Start the debugger again and connect to the target machine as described in the Setup section.
3. Type `b main` to set a breakpoint at the main function in the source code.
4. Type `continue`. This will cause the program to continue from the breakpoint set by GDB at startup.
5. Type `n`. This will cause the program to step to the next line of source code. The reason for using `n` rather than `s` or one of the instruction stepping commands is because the erroneous output indicates that the coding mistake is in the programmer's source code rather than the c library functions `atoi()` or `fprintf()`. In other words, staying within the top-level stack frame and observing the program source code step through seems like a good place to start looking for programming mistakes. Later passes through the code can be used to step into functions called from within that stack frame if the first pass proves unsuccessful.
6. Continue to type `n` until one of the program's `exit()` calls is reached, but do not actually step into that `exit()` call. Judging by the program's output above, this should bring you to the conditional block that checks the value of the local variable `n` used to store the output of `atoi()` as shown in Listing 3. Note that once execution reaches line 79 of the source code, GDB will display the output of the `fprintf()` function from like 76. This may cause display problems within the text-based UI library that GDB uses which will require the command `refresh` to fix.
   Listing 3. GDB Source Layout

```
B+ |75              if ((data.num_threads < 1) || (data.num_threads < MAX_THREAD)) {       |
   |76                      fprintf(stderr,                                                 |
   |77                              "The number of thread should between 1 and %d\n",       |
   |78                              MAX_THREAD);                                            |
  >|79                      exit(EXIT_FAILURE);                                             |
   |80              }                                                                       |
```

7. Type `p/d data→num_threads`. `p` is an alias for `print`, `/d` tells GDB to treat the expression requested as an integer in signed decimal, and `data→num_threads` is the element `num_threads` within the `struct thread_data data` data structure. This should provide the following output:

```
(gdb) p/d data->num_threads
$6 = 5
```

   Note that the integer part of "$6" will increment with each call to the gdb command `print`. The above output confirms that the argument '5' was successfully passed to the program and read into a variable to be tested, indicating that one of the logical tests for the current conditional block contains a mistake. This merits a closer look at line 75:

```
B+ |75              if ((data.num_threads < 1) || (data.num_threads < MAX_THREAD)) {       |
```

   Line 75 consists of a conditional test which is the logical OR of two arithmetic tests involving the values of `data.num_threads`, '1', and `MAX_THREAD`. The first test is true the input integer is less than 1–`(data.num_threads < 1)`. The second tests whether the input integer is less than the symbolic constant, `MAX_THREAD`–`(data.num_threads < MAX_THREAD)`. Judging by the name of this constant and the result of the test (we know it resolves to true because the value of `data.num_threads` in this case is not less than one), we can see that the comparison operator used is the culprit. The correct interpretation is that it should be '>' rather than '<'.

8. Type `kill`.

This was a simple problem to solve but the method used above could apply in any situation where source code compiles and runs without errors yet provides varied or unexpected output.

Lesson 3: Debugging With Threads

Do not fix the programming mistake found in Lesson 2. This lesson will cover the use of the `jump` command to skip blocks of code and commands specific to debugging multi-threaded programs. Before getting started, see Debugging with GDB: 5. Stopping and Starting Multi-thread Programs. TODO: upload gdb 6.8 documentation to the emac webserver then link to it here

This lesson assumes that `gdbserver` has just been started as in the Target Machine Setup above with an `ARG` value of 7.

1. Start the debugger again and connect to the target machine as described in the Setup section.
2. Type `set scheduler-locking on`. This command enables GDB to lock all threads save for the currently-selected thread from running when the `step/stepi` or `next/nexti` commands are given.
3. Set all the breakpoints that you will need for this session:
   1. Type `b main` to set a breakpoint at the main function in the source code.
   2. Type `b 75`. This will set a break point at the conditional block that checks the value of the program's single integer argument. If you recall from Lesson 2, this is the conditional which evaluates incorrectly in the modified version of the application.
   3. Type `b 143`. This will set a breakpoint in the variable assignment in the `generator` function. Careful examination of the source code will show that this function is called from a thread created by the

main thread of execution but never from the main thread itself.
   4. Type `b 129`. This will set a breakpoint in the variable assignment of the `reader` function. As with the `generator` function, any time the `reader` function is called it will be inside a thread that is not the main thread.
   5. Type `b 135`. This will set a breakpoint just after the `fflush(stdout)` statement in the `reader` function.
   6. Type `b 97`. This will set a breakpoint in the main thread after the `generator` thread has been created but before the main thread begins creating `reader` threads.
   7. Type `b 119`. This will set a breakpoint after the main thread iteratively creates the `reader` threads.
4. Optional: You may want to run the `layout split` command so that you can see both the assembly and the source code during the debug session.
5. Type `continue` then hit 'Enter' once. This will bring you to line 75 in the source code.
6. Type `j 81`. This is an alias for `jump 81` that tells GDB to have the program jump to line 81 of the source code and resume execution at the first assembly instruction represented by line 81 of the source code. This line is labeled `<main.c+196>`. *Note that the program effectively no longer checks the input it receives.*
7. Type `i th`. This will cause GDB to display a list of the application's threads currently in memory. Take a moment to consider what is happening in the program. We know that in Step 2 of this lesson we used `set scheduler-locking` to tell GDB to effectively only allow the currently-selected thread of execution to be affected by the GDB `step` and `next` commands. Others will wait at their respective breakpoints until explicitly told by the programmer to execute the next line of source code or instruction. The next breakpoint that `main` reaches occurs after the `generator` thread is created. This means that there are currently two threads of execution, the `main` thread paused at line 97 and the `generator` thread paused at line 143.
8. Type `thread 2`. This will select the `generator` thread.
9. Type `thread apply 2 n`. This will tell the `generator` thread to execute the next line of source code and pause again on the line following that. Without typing any other commands into the GDB prompt, hit 'Enter' seven more times. This should bring you to line 165 of the source code:

```
usleep(1);
```

   Notice the output of the program on the remote terminal on which `gdbserver` was run. Standard output on that terminal should show the output from the `printf()` call on line 154.

10. Type `thread 1`. This will select the `main` thread.
11. Type `continue`. This will cause the `main` thread to continue execution while `generator` remains paused at line 165. `main` pauses again at line 119, once the 7 `reader` threads have been created. Recall from step 3.4 that `b 129` set a breakpoint in the `reader` function so that the `reader` threads would pause at line 129.
12. Type `i th`. This is an alias for `info threads`. This causes GDB to print out all the threads currently in memory. Notice that there are three types of threads, `main`, `generator`, and `reader`. The `info threads` command also shows that the `reader` threads are all paused at line 129, the `generator` thread is paused at line 165, and the `main` thread is paused at line 119.
13. Type `thread 5`. This will select the third `reader` thread.
14. Type `thread apply 5 n`. Then press 'enter' seven times. This will cause the third `reader` thread to complete its task and exit gracefully using the `pthread_exit()` function.
15. Type `thread 1`. This will select the `main` thread.
16. Type `p data`. This will show the current state of the data structure that was passed to each thread. Note that each thread contains a pointer to the same data structure. This requires the use of what is known as a mutex (MUTual Exclusion) variable which is used to protect the data structure from concurrent modifications. In other words, any time the data structure is read or written to by one of the threads, they must first call the function `pthread_mutex_lock()` to ensure that no other thread currently "has" the lock. When a thread is done with the shared data structure, it calls the function `pthread_mutex_unlock()` to make the lock available to other threads. Notice that nothing about the mutex's inclusion in the data structure requires it to be used in order to read or write to the data structure. This means that any programmer who wishes to use threads to implement concurrent programming must pay close attention to

code that accesses shared data structures to ensure concurrent modifications do not occur.

17. Perform the previous four steps for as many of the `reader` threads as you want. Notice that each one prints a message to standard out providing information about the state of the shared `data` variable at the time that it has the lock. By switching to the `generator` thread once the mutex variable is unlocked, that code can be stepped through to generate a new random number for the data structure. IMPORTANT: Do not execute a line of source code containing a call to `pthread_mutex_lock()` without first ensuring that the mutex variable is unlocked. To do this, carefully perform the following steps:
    1. Type `p data→lock`. This will show the values of the mutex variable in the `data` structure variable. Make note of the value of the `owner` field.
    2. Type `i th`. This will show the current list of threads in the program. What follows is a possible output from these two commands:

```
(gdb) p data->lock
$3 = {__data = {__lock = 1, __count = 0, __owner = 1288, __kind = 0, __nusers = 1, {__spi
  __size = "\001\000\000\000\000\000\000\000\b\005\000\000\000\000\000\000\001\000\000\00
(gdb) i th
  9 Thread 1289  reader (arg=0xbeddec8c) at pthread_demo.c:129
  8 Thread 1288  reader (arg=0xbeddec8c) at pthread_demo.c:134
  7 Thread 1287  reader (arg=0xbeddec8c) at pthread_demo.c:129
  6 Thread 1286  0x00008b04 in reader (arg=0xbeddec8c) at pthread_demo.c:129
  3 Thread 1283  0x00008b04 in reader (arg=0xbeddec8c) at pthread_demo.c:129
* 2 Thread 1282  generator (arg=0xbeddec8c) at pthread_demo.c:165
  1 Thread 1281  0x00008a64 in main (argc=2, argv=0xbeddee24) at pthread_demo.c:119
(gdb)
```

    Analysis of this output requires the understanding that the third column of the `i th` output indicates that particular thread's process ID. The important part of the `p data→lock` output is the `owner` field, whose value will always either be zero or correspond to the process ID of one of the currently-running threads. In this case, the lock→__owner field clearly indicates that thread 8 currently owns the data→lock mutex variable. This would indicate that thread 8 should be stepped through until it has called the `pthread_mutex_unlock()` function before stepping into a call to `pthread_mutex_lock()` in any other function.
    To summarize, always ensure that the `owner` field of the mutex variable is equal to zero before using `pthread_mutex_lock()` while debugging. If the lock is currently owned by another thread, GDB will hang until sent an interrupt signal which will require that the entire debug process be started over.

18. The walktrhough is complete. There are two ways to end the debug session gracefully:
    - Type `monitor exit`, `kill` (confirm with 'y'), then `quit`.
    - Type `set scheduler-locking off`, `delete`, `thread 1`, `continue`, then `monitor exit`, `kill` (confirm with 'y'), `quit`. This will allow the program to finish executing before ending the session.

GNU GDB Documentation

Again, for more information on how to debug with GDB, refer to the GDB Manual (http://sourceware.org/gdb/current/onlinedocs/gdb.html) . This is a valuable resource for anyone just learning to debug software and will go into much greater detail than is possible in this guide.

Next Steps

If you have not done so already, be sure to check out the EMAC OE SDK Example Projects or learn to create your own New Project.

Also, give the EMAC Eclipse IDE a try. Sometimes it is simpler to use an IDE for large projects, especially with the ability to automate the makefile creation process.

See Also

- EMAC Software Development Kit
  - Install EMAC OE SDK
  - Configure EMAC OE SDK
  - Example Projects
  - New Project
  - Debugging With gdbserver

» emac_oe_gadget » time » emac_oe_development » esdk » install » configure » example » new » linux_start » debug

---