Using GDB with Nios® V

For Quartus® Prime 23.1

1 Introduction

This tutorial provides instructions for using the *GNU Project Debugger* (GDB) with the *Nios® V* processor, to develop and debug software programs written in assembly-language or C code. We assume that you are running the software tools described in this tutorial in the DESL laboratory rooms on the University of Toronto campus, and therefore all of the required tools are already installed on the computer system. If you are using this tutorial at home on your own computer, then you may wish to refer to the more complete version of the tutorial that is available in the *Computer Organization System Design* section of the FPGAcademy.org website. That version includes a section about installing the required software and hardware tools.

This tutorial covers the following topics:

- Obtaining the design examples used in the tutorial
- Configuring the DE1-SoC board
- Developing and debugging Nios V assembly-language programs
- Developing and debugging C programs
- GDB command reference

2 Developing and Debugging Nios V Assembly-Language Programs

We will first present a few examples that show how to use GDB to develop and debug Nios V assembly-language code. You will likely not be familiar with some of the Nios V features that are used in the more *advanced* examples, until the associated topics have been covered in course lectures. However, you can still read through all of the examples to become familiar with the content, and then revisit the tutorial as your course lectures progress. This tutorial also includes example designs that use C code, starting in Section 3.

2.1 Installing the Tutorial Design Examples

Along with this tutorial, you have been provided with *Design Files* that are used to illustrate various features of the *GDB* software for developing and debugging Nios V programs. Download to your computer the provided *Using_GDB_Nios_V_design_files.zip* file. Then, uncompress this archive into any folder of your choice. We will refer to the examples of code and other files in this folder throughout the tutorial. Figure 1 shows the folders included in the *design files*, assuming that they have been installed into a folder named `GDB_tutorial`.

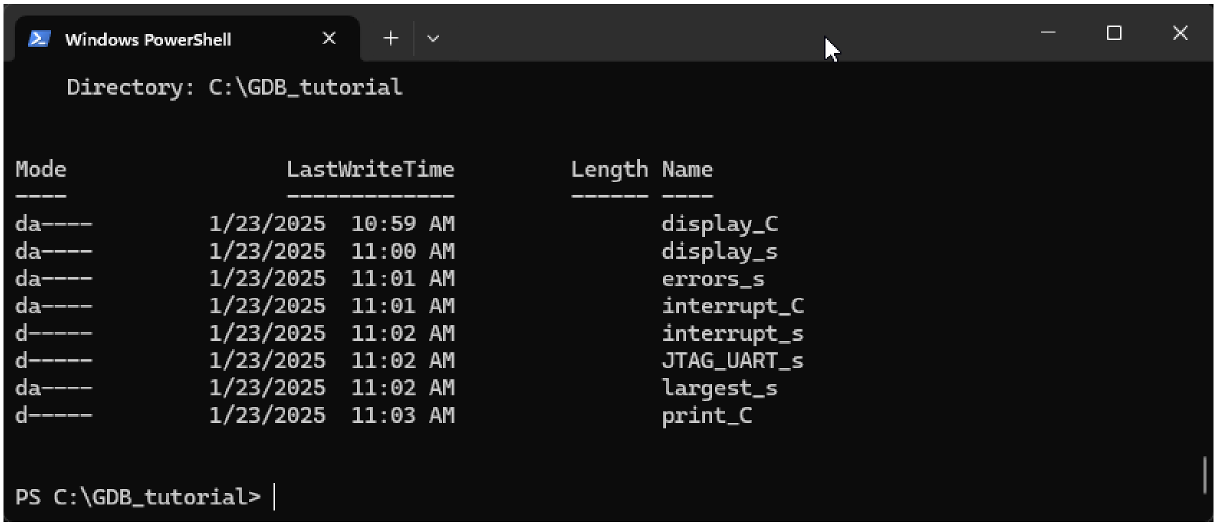


Figure 1. The *Design Files* folders.

2.2 Using the GNU Make Program

In this tutorial all tools are executed by using the command-line environment provided by *Windows PowerShell*. Open a *PowerShell* terminal using a method of your choosing. Then, navigate to the *design files* folder called `C:\GDB_tutorial\largest_s`. As illustrated in Figure 2, this folder contains an example of a Nios V assembly-language program, *largest.s*, an executable *batch* file *gmake.bat*, and a *Makefile*.

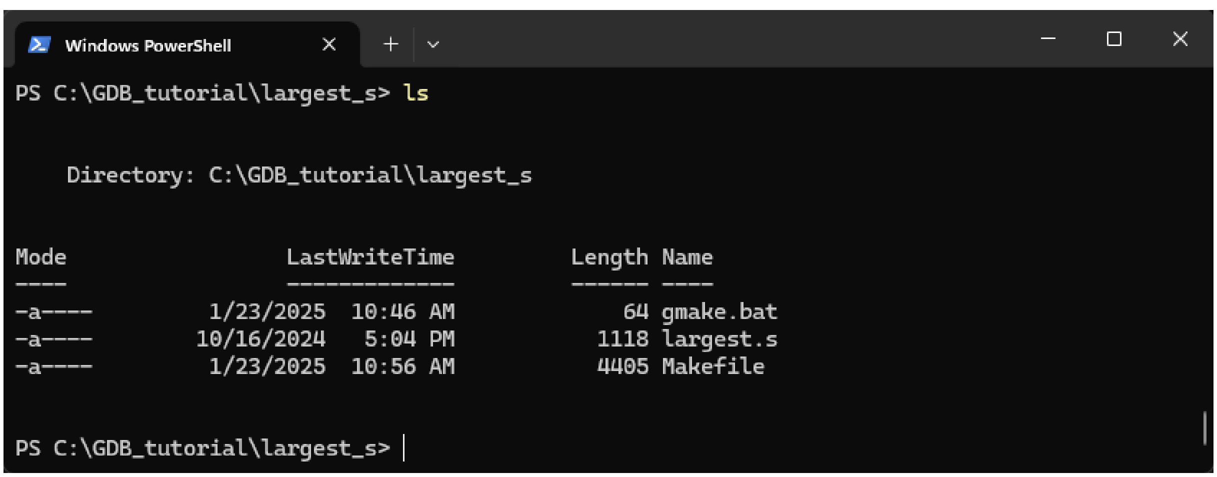


Figure 2. The contents of the largest_s design files folder.

In this tutorial the tools that we use for developing and debugging Nios V programs are executed via the *GNU make* program. A copy of *GNU make* is included on the DESL computers in the location

```
C:/intelFPGA/QUARTUS_Lite_V23.1/fpgacademy/AMP/bin/make.exe
```

To make sure that you execute this required version of *make.exe*, rather than some other *make.exe* executable that could also be installed on the DESL computers, we provide *gmake.bat*. You will run the *make* program via this batch script, which includes the full path given above to *make.exe*.

In the folder of Figure 2 open the *Makefile* in any text editor of your choice. The first few lines of this file are displayed in Figure 3. Line 1 defines a variable called `INSTALL` that specifies the folder in which the software needed for this tutorial has been installed. The setting given in the figure matches the installation folder used for the computers in the DESL lab.

```
1 INSTALL := C:/intelFPGA/QUARTUS_Lite_V23.1
2
3 MAIN    := largest.s
4 HDRS    :=
5 SRCS    := $(MAIN)
6
7 SHELL   := cmd.exe
8
9 # DE1-SoC
10 JTAG_INDEX_SoC  := 2
11
```

Figure 3. The first few lines of the *Makefile*.

2.3 Configuring the DE1-SoC Board

Connect a DE1-SoC board to your computer. The board should be connected by plugging a cable that has a *Type-A USB* connector into the *USB Blaster* port on the board and connecting the other end of this cable to any *USB* port on your computer. Ensure that the DE1-SoC board is properly powered on. To configure your DE1-SoC board with the desired Nios V computer system, in the terminal window of Figure 2 execute the command:

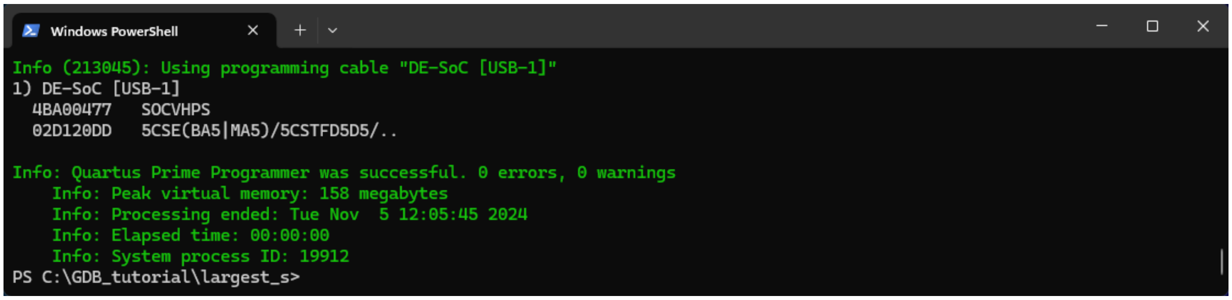
```
./gmake DE1-SoC
```

Note that the "`./`" above is used to specifies a file in the current working folder. This command uses GNU *make* to run the Quartus *Programmer*, which configures the DE1-SoC board with the *DE1-SoC Computer with Nios V* system. If the command completes without errors, then you can skip ahead to Section 2.4 and begin using GDB with Nios V. Note that in some cases it can take several seconds, or more, to configure the board.

If the Quartus *Programmer* fails to configure your DE1-SoC board, then try running the command:

```
./gmake DETECT_DEVICES
```

This command checks which devices are visible on the *USB Blaster* cable that is connected to your computer. Part of the expected output from this command is displayed in Figure 4. It shows two devices being detected: first an *SOCVHPS* device, followed by a Cyclone V *5CSE FPGA* device. If the output produced from your board shows these two devices, but in the opposite order, then you have to modify your *Makefile*. Change the variable `JTAG_INDEX_SoC` shown in Line 10 of Figure 3 from the value `2` to the value `1`. You should now be able to successfully configure your DE1-SoC board by executing the `./gmake DE1-SoC` command.



```
Windows PowerShell
Info (213045): Using programming cable "DE-SoC [USB-1]"
1) DE-SoC [USB-1]
  4BA00477   SOCVHPS
  02D120DD   5CSE(BA5|MA5)/5CSTFD5D5/..

Info: Quartus Prime Programmer was successful. 0 errors, 0 warnings
    Info: Peak virtual memory: 158 megabytes
    Info: Processing ended: Tue Nov  5 12:05:45 2024
    Info: Elapsed time: 00:00:00
    Info: System process ID: 19912
PS C:\GDB_tutorial\largest_s>
```

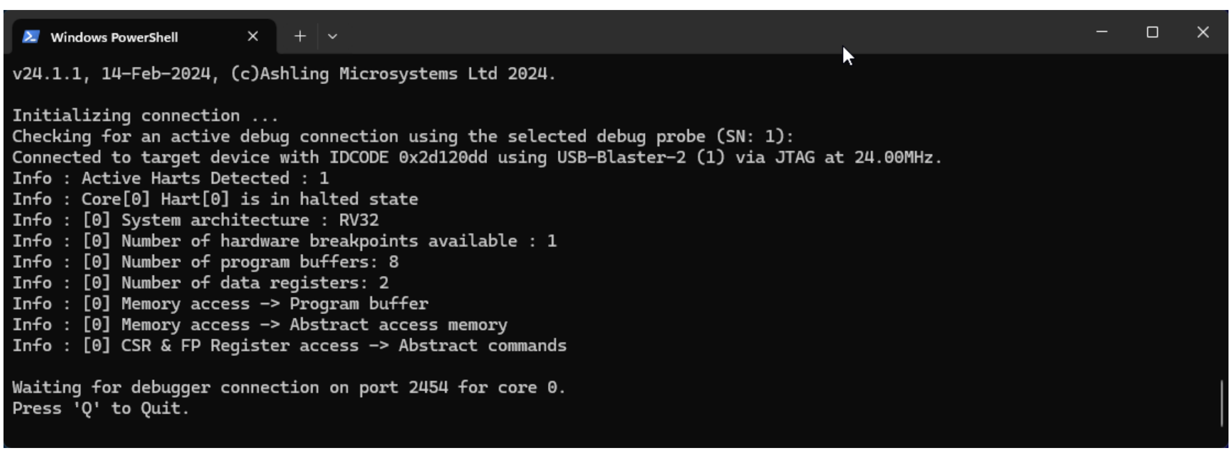
Figure 4. The output from `./gmake DETECT_DEVICES`.

2.4 Using the GDB Server and Client

To develop Nios V programs, you need to use **two** PowerShell terminals: the first one is used to open the *GDB Server*, and the second one is used run the *GDB Client*. To start the GDB Server, in the terminal window of Figure 2 execute the command:

```
./gmake GDB_SERVER
```

The server will then remain running in this window, as indicated in Figure 5. Now, open a second PowerShell window (in this tutorial we show screen captures using Microsoft Windows 11, in which you can click on the + button near the top of Figure 5 to open a new PowerShell *tab*. But in the DESL labs you are running Windows 10, where you have to open a new PowerShell window by using the start menu or some other method). In this *second* PowerShell terminal navigate again to the same folder as in Figure 2.



```
Windows PowerShell
v24.1.1, 14-Feb-2024, (c)Ashling Microsystems Ltd 2024.

Initializing connection ...
Checking for an active debug connection using the selected debug probe (SN: 1):
Connected to target device with IDCODE 0x2d120dd using USB-Blaster-2 (1) via JTAG at 24.00MHz.
Info : Active Harts Detected : 1
Info : Core[0] Hart[0] is in halted state
Info : [0] System architecture : RV32
Info : [0] Number of hardware breakpoints available : 1
Info : [0] Number of program buffers: 8
Info : [0] Number of data registers: 2
Info : [0] Memory access -> Program buffer
Info : [0] Memory access -> Abstract access memory
Info : [0] CSR & FP Register access -> Abstract commands

Waiting for debugger connection on port 2454 for core 0.
Press 'Q' to Quit.
```

Figure 5. Running the GDB Server.

This part of the tutorial uses an assembly-language program, *largest.s*, which is shown in Figure 6. This program searches through a list of integers that is stored in memory and finds the largest number in the list. Assemble this program (using the *second* PowerShell window that you opened) by executing the command `./gmake COMPILE`.

You could also just type `./gmake`, because `COMPILE` is the first target in the *Makefile*. As illustrated in Figure 7, this command runs the Nios V assembler and linker tools to generate the Nios V executable file *largest.elf*.

```
# Program that finds the largest number in a list of integers
.global _start
_start:     la      t0, result      # t0 = pointer to the result
            lw      t1, 4(t0)       # t1 = counter, initialized with N
            addi    t2, t0, 8       # t2 = pointer to the first number
            lw      t3, (t2)        # t3 = largest found so far
loop:       addi    t1, t1, -1      # decrement counter
            beqz    t1, done        # done when counter is 0
            addi    t2, t2, 4       # point to the next number
            lw      t4, (t2)        # get the next number
            bge     t3, t4, loop    # compare to largest found
            mv      t3, t4          # remember new largest
            j       loop

done:       sw      t3, (t0)        # store result
stop:       j       stop            # wait here

result:     .word   0               # result will be stored here
N:          .word   7               # number of entries in the list
numbers:    .word   4, 5, 3, 6      # numbers in the list
            .word   1, 8, 2         # ...
```

Figure 6. A program that finds the largest number in a list.

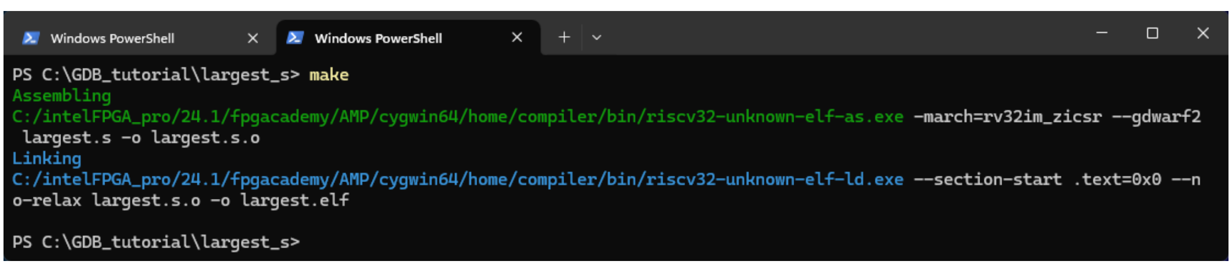


Figure 7. Making the executable file *largest.elf*.

Now you can run the *GDB Client* by executing the command:

```
./gmake GDB_CLIENT
```

The GDB Client will connect to your DE1-SoC board, load the executable file *largest.elf*, initialize some Nios V control registers, and set the Nios V program counter register, *pc*, to the start of the program. The output produced by this command is displayed in Figure 8.

```
PS C:\GDB_tutorial\largest_s> make GDB_CLIENT
riscv32-unknown-elf-gdb.exe -silent -ex "target remote:2454" -ex "set $mstatus=0" -ex "set $mtvec=0" -ex "set $mie=0" -e
x "load" -ex "set $pc=_start" -ex "info reg $pc" "largest.elf"
Reading symbols from largest.elf...
Remote debugging using :2454
_start () at largest.s:3
3       _start:     la      t0, result      # t0 = pointer to the result
Loading section .text, size 0x5c lma 0x0
Start address 0x00000000, load size 92
Transfer rate: 5 KB/sec, 92 bytes/write.
pc             0x0      0x0 <_start>
(gdb) |
```

Figure 8. Starting the GDB Client.

We will use the *largest.s* program as an example to illustrate some basic GDB commands. A summary of the GDB commands used in this tutorial is provided in Appendix A. Of course, a lot of documentation about GDB commands can also be found on the Internet.

In the GDB Client type the `list` command, as shown in the Figure 9, to see the loaded program.

```
(gdb) l
1       # Program that finds the largest number in a list of integers
2       .global _start
3       _start:     la      t0, result      # t0 = pointer to the result
4                   lw      t1, 4(t0)       # t1 = counter, initialized with N
5                   addi    t2, t0, 8       # t2 = pointer to the first number
6                   lw      t3, (t2)        # t3 = largest found so far
7       loop:       addi    t1, t1, -1      # decrement counter
8                   beqz    t1, done        # done when counter is 0
9                   addi    t2, t2, 4       # point to the next number
10                  lw      t4, (t2)        # get the next number
(gdb)
```

Figure 9. The output of the `list` command.

Next, execute the first two instructions in the program by using the GDB `step` command twice. Then, execute the commands `info reg t0` and `info reg t1` to see that register `t0` holds the address in memory of the `result` label, which is `0x38`, and that register `t1` has the number of elements in the list, which is `7` (this value is specified at the label `N` in the code in Figure 6). The results of these commands are displayed in Figure 10. You can see the contents of memory by using the `x` command. Enter `x/4x result` to see the four words of memory starting at the address of the `result` label (`/4x` designates four `words` displayed in `hexadecimal`).

As illustrated in Figure 11 set a breakpoint at line 7 in the source code, which corresponds to the label `loop`, by using the command `break 7`. Then, run to this breakpoint twice by using the `continue` command. Check the value of register `t3` to see that the largest number found in the list so far is `5`. Now, clear the breakpoint by using the command `clear 7`. The program ends with an infinite loop at the label `stop`, as seen in Figure 6. Set a breakpoint at this label by using the command `break stop`. Enter `continue` to resume the program until it stops at the breakpoint. Finally, use the command `info reg t3` to see that the program found the largest number in the list, which is `8`, and enter `x/4x result` to see that this result has been stored into memory.

We are now finished with the *largest_s* example. As demonstrated in Figure 12, disconnect from your DE1-SoC board by executing the `detach` command. Finally, execute the `quit` command. If you see the prompt `Terminate batch job (Y/N)?` respond with `n`.

In the GDB Server terminal of Figure 5, type `q` to quit. While it is not absolutely necessary to exit the server before starting to work on another Nios V program, it is a good idea to do so. The server occasionally experiences communications failures with the DE1-SoC board and then has to be restarted—hence, leaving the server running for long periods of time may not be a good approach.

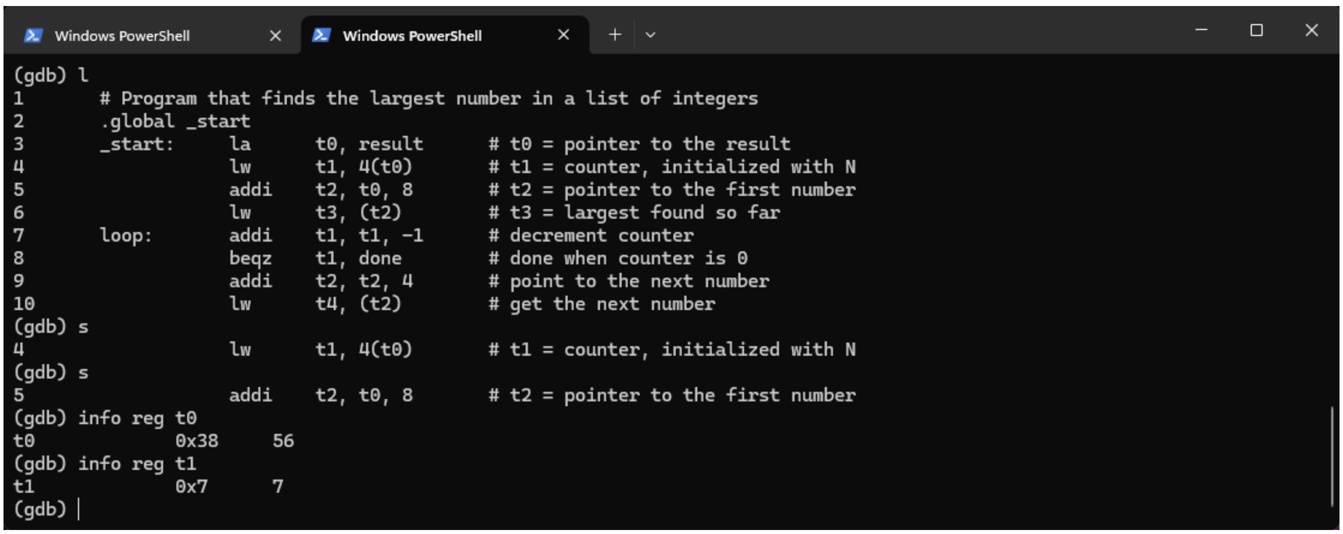


Figure 10. Executing a few GDB commands.

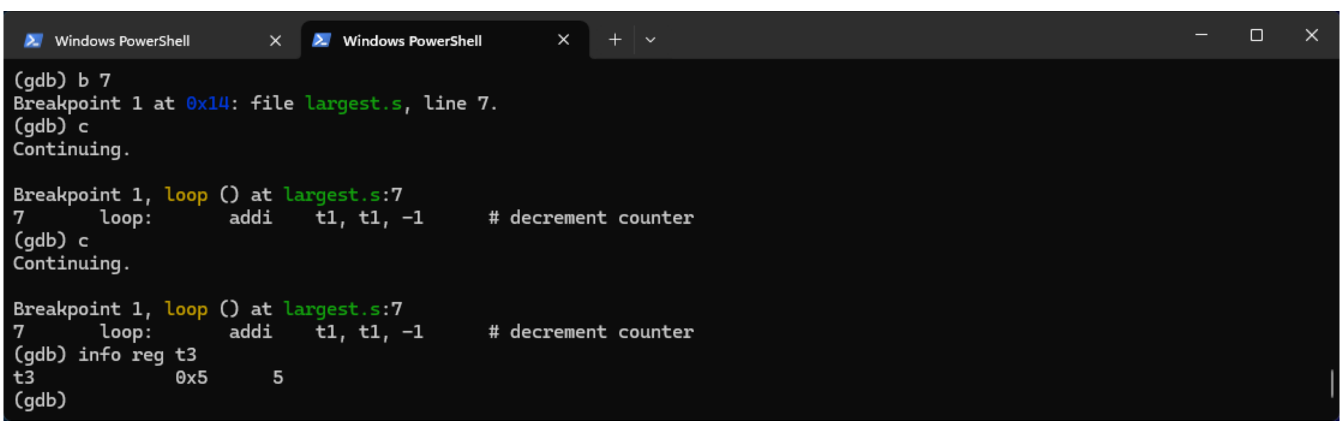


Figure 11. Using a breakpoint.

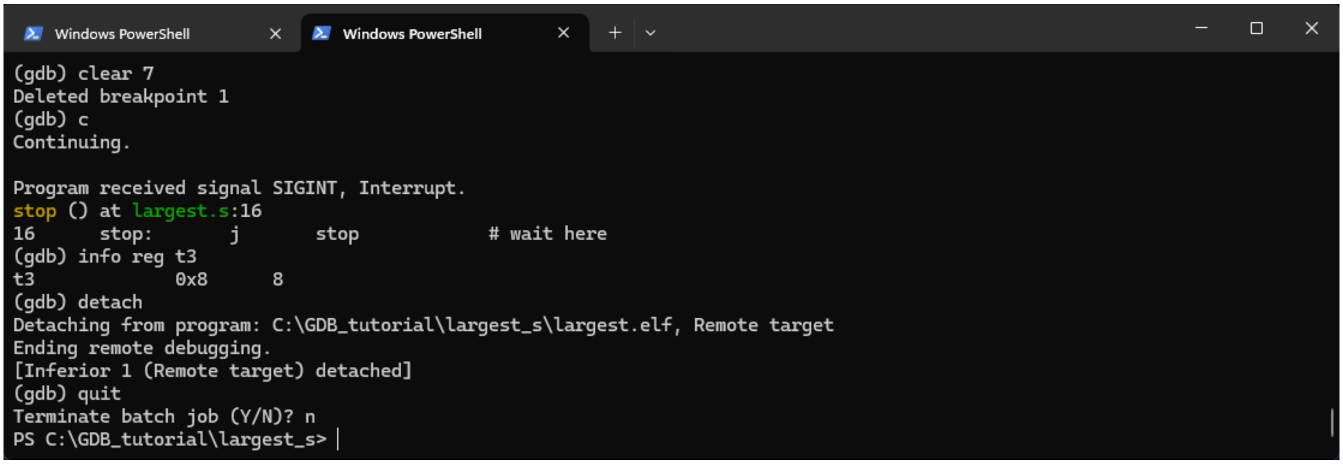


Figure 12. Completing the program and quitting from GDB.

2.5 Setting up Your Own Assembly-Code Makefile

It is easy to customize the *Makefile* shown in Figure 3 so that you can use it with any assembly-language code of your choosing. `Line 3` of the *Makefile* has to specify the name of the assembly-code file that has the `_start` label, which designates the beginning of the program. Any `.s` header files that are used with the program can be listed in `Line 4`. Finally, any additional assembly-language source-code files can be listed in `Line 5`.

Having learned the basics about using GDB with Nios V in Section 2.4, we will now utilize the various design files examples provided with this tutorial to illustrate additional GDB capabilities.

2.6 Using Simple I/O Devices with Assembly Code

When starting to work on a new design example it is a good approach to power your DE1-SoC board `off`, and then `on` again, so that the system is reset. Open a PowerShell terminal and navigate to the folder for the `display_s` design example. In this folder, execute the command `./gmake DE1-SoC` to configure your board. If this programming step fails, refer to the discussion in Section 2.3 for suggestions as to how to fix any issues. Once your board is successfully configured, execute the command `./gmake GDB_SERVER`. Now, open a second PowerShell tab (as described in Section 2.4) and, as illustrated in Figure 13, navigate again to the `display_s` folder and execute the command `./gmake COMPILE`, followed by `./gmake GDB_CLIENT`.

Within the GDB Client, use `list 1,14` to see the source-code of the program. As shown in Figure 14, the program first sets up three pointers to I/O devices in the *DE1-SoC Computer*: register *t0* is initialized to the address of the *LEDR* red light port, register *t1* to the address of the *SW* slide-switch port, and register *t2* to the address of the port connected to 7-segment display *HEX3* to *HEX0*. The program then executes an endless loop in which it loads the current value of the *SW* switch port and stores this value to both the *LEDR* and *HEX0* display ports.

Enter the command `break loop`, and then use `continue` to stop the program at this breakpoint. Now, run through iterations of the loop in the program for a while by executing the command `continue 150`. This com-

```
Windows PowerShell    Windows PowerShell
PS C:\GDB_tutorial\display_s> make
Assembling
C:/intelFPGA_pro/24.1/fpgacademy/AMP/cygwin64/home/compiler/bin/riscv32-unknown-elf-as.exe -march=rv32im_zicsr --gdwarf2
 display.s -o display.s.o
Linking
C:/intelFPGA_pro/24.1/fpgacademy/AMP/cygwin64/home/compiler/bin/riscv32-unknown-elf-ld.exe --section-start .text=0x0 --n
o-relax display.s.o -o display.elf

PS C:\GDB_tutorial\display_s> make GDB_CLIENT
riscv32-unknown-elf-gdb.exe -silent -ex "target remote:2454" -ex "set $mstatus=0" -ex "set $mtvec=0" -ex "set $mie=0" -e
x "load" -ex "set $pc=_start" -ex "info reg $pc" "display.elf"
Reading symbols from display.elf...
Remote debugging using :2454
_start () at display.s:6
6       _start: li      t0, LEDR_BASE       # pointer to LEDR port
Loading section .text, size 0x24 lma 0x0
Start address 0x00000000, load size 36
Transfer rate: 2 KB/sec, 36 bytes/write.
pc             0x0      0x0 <_start>
(gdb) |
```

Figure 13. Starting GDB for the `display_s` example.

mand runs the program until the breakpoint at `loop` has been encountered 150 times. While the program is running, try different settings on the SW_{6-0} switches on the DE1-SoC board and observe the *LEDR* lights and *HEX0* display.

Wait until the program has stopped executing and control has been returned to the GDB Client. Now, set a new pattern of your choice on the *SW* switches and then enter the `step` command, followed by `info reg t3` to see the value loaded from the *SW* port. Execute `step` again and observe the *LEDR* lights, then use `step` a third time and observe the *HEX0* display.

```
Windows PowerShell    Windows PowerShell
(gdb) l 1,14
1       # Program that displays SW switch settings on LEDR and HEX0
2
3       .include "address_map.s"
4
5       .global _start
6       _start: li      t0, LEDR_BASE       # pointer to LEDR port
7               li      t1, SW_BASE         # pointer to SW port
8               li      t2, HEX3_HEX0_BASE  # pointer to HEX port
9
10      loop:   lw      t3, (t1)            # read from SW
11              sw      t3, (t0)            # write to LEDR
12              sw      t3, (t2)            # write to HEX0
13              j       loop
14
(gdb) |
```

Figure 14. The `list` command.

Use `continue` again to get to the top of the loop, and then `step` to execute the *load* instruction at `Line 10`. Now, use the GDB command `set $t3 = 0x3ff` to overwrite the value loaded into register *t3*. Use `step` again and then observe on the DE1-SoC board that the value you placed into register *t3* turns on all ten *LEDR* lights.

We are now done with this design example, so use the `detach` command to disconnect from the DE1-SoC board. Finally, `quit` from the GDB Client, and then go to the GDB Server PowerShell tab and type `q` to close the server.

2.7 Using Interrupts with Assembly Code

The assembly code example for this part of the tutorial displays a one-second binary counter on the red lights *LEDR*. The speed of the counter is controlled by using interrupts from the Nios V Machine Timer.

Open a PowerShell terminal and navigate to the `interrupt_s` folder. If not already done, configure your DE1-SoC board by running `./gmake DE1-SoC`. Execute `./gmake GDB_SERVER`. In a second PowerShell tab navigate again to the `interrupt_s` folder and run `./gmake COMPILE` and `./gmake GDB_CLIENT`.

In the GDB Client, as illustrated in Figure 15, run `list 1,15`. Use `step` to execute the instruction on `Line 5` that initializes the stack pointer register. Next, enter `x/4x 0xff202100`. This command displays the contents of the memory-mapped 64-bit Nios V `Machine Timer` registers, which are referred to as *mtime*, which has the address `0xff202100`, and *mtimecmp*, which has the address `0xff202108`.

Next, set a breakpoint using `break 9`. Then, execute `continue`, which runs the Nios V program to call the subroutine *set_timer* and then stops, after returning, at the breakpoint on `Line 9`. Again, as illustrated in Figure 16, use `x/4x 0xff202100` to display the Machine Timer registers. As indicated in the figure, the *mtime* register was cleared to 0 (and then continued counting up at its 100 MHz clock rate), and the *mtimecmp* register was set to 100,000,000 (`0x5f5e100`) to provide machine timer timeouts for every one second.

```
Windows PowerShell    Windows PowerShell
1       # Program that displays a binary counter on LEDR
2       .include "address_map_niosv.s"
3
4       .global _start
5       _start:     li      sp, 0x20000       # initialize the stack location
6                   jal     set_timer         # initialize the timer
7
8                   # Set handler address, enable interrupts
9                   la      t0, handler
10                  csrw    mtvec, t0         # set trap address
11                  li      t0, 0b10000000    # set the enable pattern
12                  csrs    mie, t0           # enable timer interrupts
13                  csrsi   mstatus, 0x8      # enable global interrupts
14
15                  la      s0, counter       # pointer to counter
(gdb) |
```

Figure 15. The `interrupt_s` example.

The next five instructions in the program set up Nios V interrupts as needed for this example. As shown in Figure 17, display the contents of the (uninitialized) *mtvec* control register by using `info reg mtvec`. Enter `step 2` to execute two instructions, and then use `info reg mtvec` to see that this register has been initialized to `0x68`. Enter `info symbol 0x68` to see that this is the address in memory of the interrupt *handler* routine. Display the current contents of the *mie* and *mstatus* control registers with `info reg mie status`. Execute three more instructions (`step 3`) and then enter `info reg mie mstatus` again to see that the *mie* register now contains the value `0x80`, which has the interrupt-enable bit corresponding to the Machine Timer set to 1, and *mstatus* shows

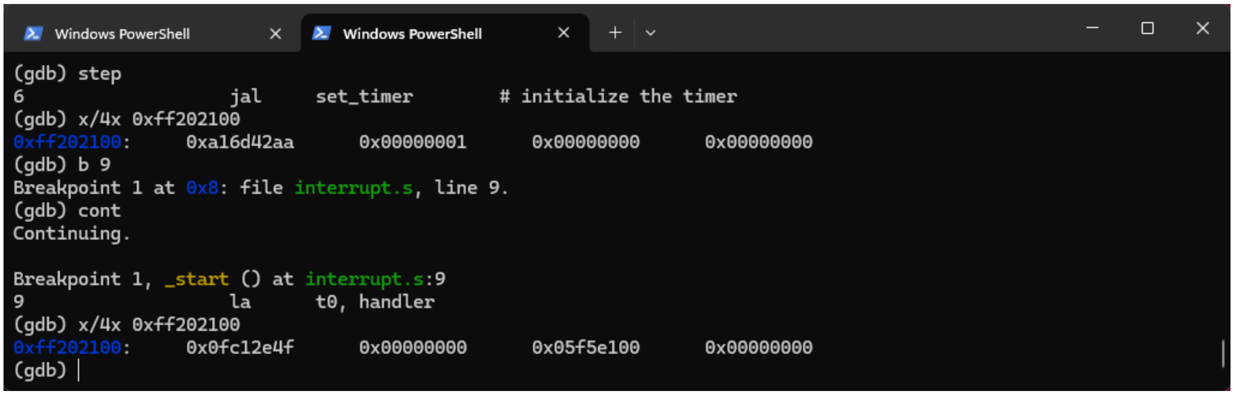


Figure 16. Examining the Nios V Machine Timer registers.

that the *Machine-mode Interrupt Enable* bit (*MIE*) in this register is now set to 1, meaning that Nios V interrupts are now enabled (*Machine* mode is the only processor mode supported in Nios V).

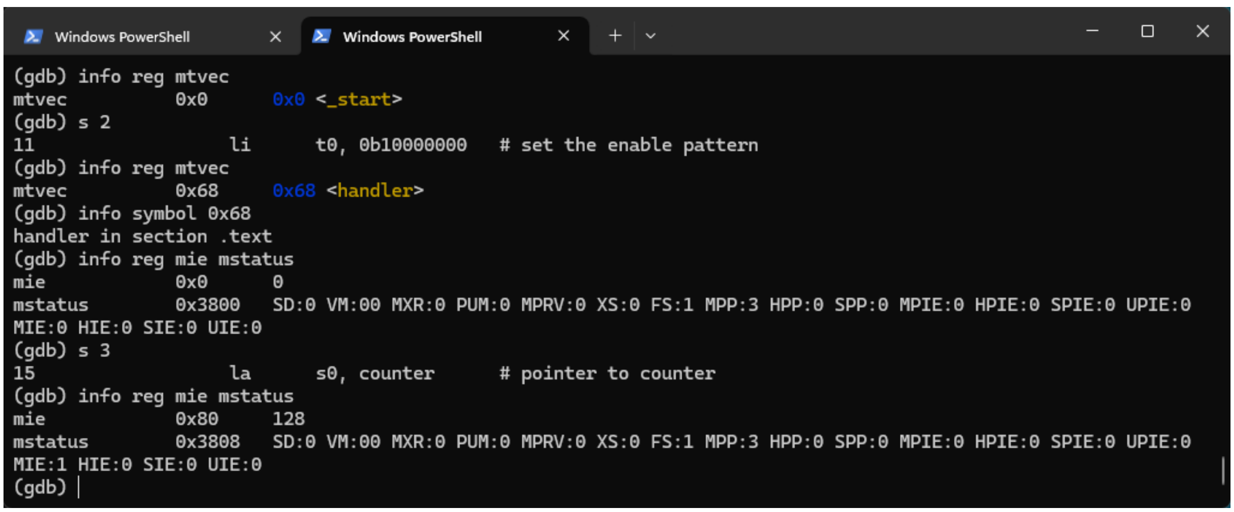


Figure 17. Examining Nios V control registers.

Clear the breakpoint that was previously set by entering `clear 9`. Enter the command `continue &`. The `&` at the end of this command means "run the program in the *background*," and immediately provide the GDB command prompt so that commands can still be executed. While the Nios V program is running, observe that the *LEDR* lights show a binary counter incrementing once per second. Now, stop the running program by executing the GDB command `interrupt`. GDB will stop the program, in the same manner as when a breakpoint is reached.

Next, enter `break handler` to set a breakpoint at the interrupt handler routine. Note that this code is in a different source-code file, *handler.s*, from the main program. Enter `continue` to run the program until it reaches the breakpoint. Enter `list` to see the first few lines of code in the *handler* routine, as displayed in Figure 18. Execute the five instructions displayed in the figure, using `step 5`. Then, enter `info reg t0` to see the cause of the interrupt (contents of the *mcause* control register). The displayed value `0x80000007`, as seen in Figure 19, shows that a hardware interrupt has occurred (`0x8`) from the device with interrupt number `7`. This is the expected result, as interrupt 7 corresponds to the Machine Timer.

Clear the handler interrupt by entering `clear handler`, and then execute `continue &` to run the program in the *background*. Observe the *LEDR* lights to get an indication of the value of the binary *counter* being displayed. Enter `interrupt` to stop the program and return control to the GDB Client. To see the current value of the *counter* use the command `x counter`. It is possible to change the value of this "variable" by using the `set` command. For example, set the counter to the value `0x3f0` by using `set {int} counter = 0x3f0`. The cast to type `{int}` is required so that GDB knows the *type* of the variable. Enter continue and observe the new value of the counter displayed on the *LEDR* lights. Another way to modify the value of the counter is to first find its address with the command `info address counter`. This command returns the address value `0x64`. Thus, an alternative way to set the counter to the value `0x3f0` is to used the command `set *0x64 = 0x3f0`. Here `*0x64` uses the syntax of the C language to set the *contents* of an address (*pointer*).

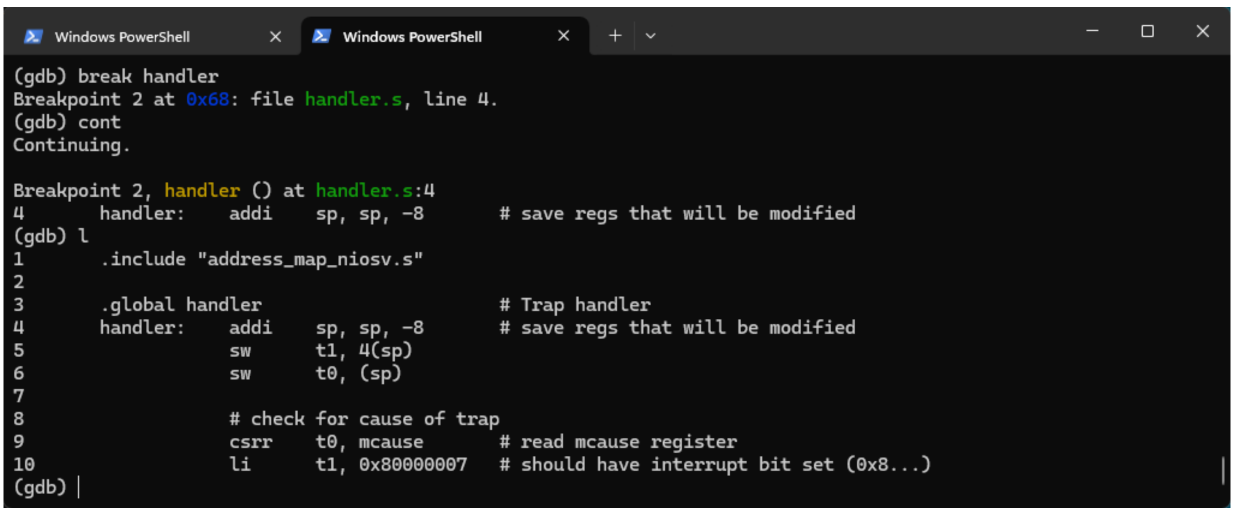


Figure 18. The interrupt *handler* routine.

We are now finished with this example, so enter `detach` to close the connection between the GDB Client and the DE1-SoC board, and then enter `quit`.

2.8 Using a Terminal to Print Text Messages from Assembly Code

In this example we will show how you can use assembly code to "print" text messages to a terminal window. This example can be found in the design files folder `JTAG_UART_s`. As done previously, open two PowerShell terminals and navigate to the proper folder. In one terminal start the GDB Server. In the other terminal, execute `./gmake` to build the program's executable file, and then start the GDB Client. Now, open a *third* PowerShell terminal and navigate again to the `JTAG_UART_s` folder. Execute the command `./gmake TERMINAL`. This command creates a communications link between the PowerShell terminal and the JTAG UART on the DE1-SoC board, which can be used to display text messages.

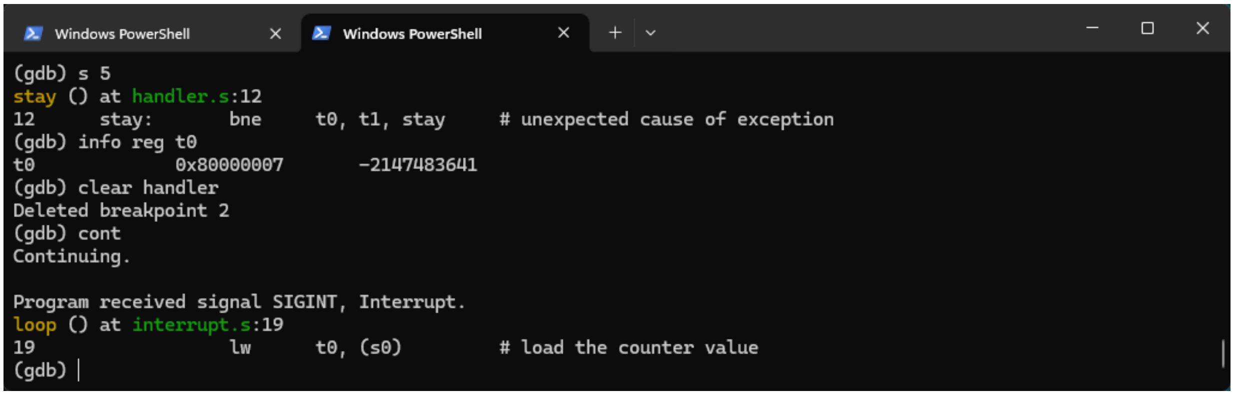


Figure 19. Checking the cause of the interrupt.

In the GDB Client run the assembly program by entering `continue &` to run the program in the *background*. Observe that the message

```
JTAG UART example code
```

appears in the terminal that is connected to the JTAG UART. Also, on a separate line the > prompt is shown. Click on this line, and then type some text with your keyboard. The text is simply echoed back to the terminal window by the assembly program that is running.

In the GDB Client enter `interrupt` to stop the running program. Then, to see how GDB can be used to restart a program enter the command `set $pc = _start`, or (equivalently) `set $pc = 0`. Use `continue &` to restart the program in the *background*, and observe the JTAG terminal window.

Again, enter the `interrupt` command to stop the program. Then, `detach` from the GDB Client and `quit`, and also quit from the GDB Server. Finally, close the connection to the JTAG terminal by typing `^C` in its window (while holding down the `ctrl` keyboard key, press `C`).

2.9 Using a Trap Handler to Catch Exceptions

As a final example using assembly language, we will show how you can handle certain error conditions that may arise in assembly code. This example can be found in the design files folder `errors_s`. As done previously, open two PowerShell terminals and navigate to the proper folder. In one terminal start the GDB Server. In the other terminal, execute `./gmake` to build the program's executable file, and then start the GDB Client.

In case an assembly program causes a Nios V error condition, such as a *misaligned* address, we can include in our assembly-code program a *trap handler* that catches such an error. In the GDB Client enter `list 1,12` as displayed in Figure 20. After first initializing the stack pointer *sp*, the program sets the Nios V *mtvec* control register to the address of a *trap handler* routine. Then, in `Line 6` the program initializes register *t0* to point to a data word in memory, and then in `Line 7` loads that word into register *t1*.

Enter the step command a few times to reach `Line 8`. Enter `step` again to execute this instruction, which increments the value in register *t0*, so that it is no longer a multiple of four (*not* word aligned). Now, enter `step &` to execute, in the *background*, the instruction on `Line 9`. Since the address value in *t0* is not word-aligned, this `lw` instruction causes a Nios V exception and Nios V transfers control to the address in the *mtvec* register, which is (*handler*). The handler code is displayed in Figure 21.

```
pc             0x0      0x0 <_start>
(gdb) l 1,12
1       .global _start
2       _start:     li      sp, 0x20000      # initialize the stack location
3                   la      t0, handler      # Set handler address
4                   csrw    mtvec, t0        # set trap address
5
6                   la      t0, data_word
7                   lw      t1, (t0)         # this load will work
8                   addi    t0, t0, 1
9                   lw      t1, (t0)         # this load will cause a trap
10      stop:       j       stop
11
12      data_word:  .word   0xa5a5a5a5       # example data
(gdb)
```

Figure 20. A program that causes an exception.

```
handler:    addi    sp, sp, -4        # save regs that will be modified
            sw      t0, (sp)

            csrr    t0, mcause        # cause of the trap
stay:       bnez    t0, stay          # stay here to allow inspection of exception
                                      # mepc points to the offending instruction
                                      # mtval has the offending address

            lw      t0, (sp)          # restore regs
            addi    sp, sp, 4
            mret
```

Figure 21. The handler routine.

As a result of the exception, the program will be caught in the loop at the label `stay`. Execute the `interrupt` command to return control to the GDB Client. Then, as depicted in Figure 22, execute the command `info reg mcause mepc mtval`. The *mcause* register has the value `4` because this is the exception code that indicates an *address alignment* error. The *mepc* register has the address of the instruction that caused this error, which is `0x20` (`Line 9` in Figure 20), and the *mtval* register shows the value of the offending address, which is `0x29`.

Similar exception-handler code as demonstrated in this example can be included in any assembly-language program, so that inadvertent errors that cause Nios V exceptions can be caught and examined.

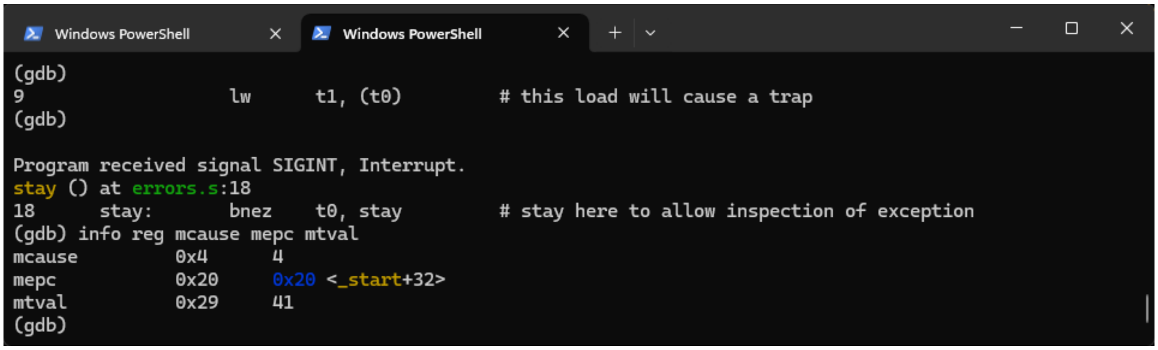


Figure 22. Examining the cause of the exception.

3 Developing and Debugging Nios V C-Language Programs

This section provides examples of using GDB for Nios V with C code. The process is mostly the same as for using assembly-language code, but the *Makefile* and some GDB commands are somewhat different.

3.1 Using I/O Devices with C Code

In this part of the tutorial we will use GDB to run a C program that accesses some simple I/O devices. As in previous examples, open two PowerShell terminals. In each terminal navigate to the folder for this design example, which is `display_C`. Use one terminal to start the GDB server. In the other terminal first execute `./gmake`, which builds the executable program by running the C compiler and linker, and then start the GDB Client, as shown in Figure 23.

Within the GDB Client enter `break main`, and then run the program using `continue`. Type `list` to see the beginning part of the main program for this example, as displayed in Figure 24.

Enter `break 15` to set a breakpoint, and then use `continue` to run to Line 15 in the source code. On the DE1-SoC board, set the *SW* switches to any value of your choosing, for example `0x7f`. Use `step` to execute the C statement on Line 15, and then execute `print /x value` to examine the data that was loaded, as depicted in Figure 25. To see which Nios V register is used to hold this data, execute the `disassemble` command. As illustrated in Figure 26, register *a5* is used to hold this *value*. Display the contents of this register by entering `info reg a5`, as seen in Figure 27. Use `step` to execute the next instruction, and observe that the *LEDR* lights are updated. Then, enter `continue` to run the program until it reaches the breakpoint again at Line 15. Observe that the *HEX0* display is now updated.

```
PS C:\GDB_tutorial\display_C> make
Compiling
C:/intelFPGA_pro/24.1/fpgacademy/AMP/cygwin64/home/compiler/bin/riscv32-unknown-elf-gcc.exe -Wall -c -g -O1 -ffunction-s
ections -fverbose-asm -fno-inline -gdwarf-2  -march=rv32im_zicsr -mabi=ilp32 display.c -o display.c.o
Linking
C:/intelFPGA_pro/24.1/fpgacademy/AMP/cygwin64/home/compiler/bin/riscv32-unknown-elf-gcc.exe -Wl,--defsym=__stack_pointer
$=0x4000000 -Wl,--defsym  -Wl,JTAG_UART_BASE=0xff201000 -lm -march=rv32im_zicsr -mabi=ilp32  display.c.o -o display.elf

PS C:\GDB_tutorial\display_C> make GDB_CLIENT
riscv32-unknown-elf-gdb.exe -silent -ex "target remote:2454" -ex "set $mstatus=0" -ex "set $mtvec=0" -ex "set $mie=0" -e
x "load" -ex "set $pc=_start" -ex "info reg $pc" "display.elf"
Reading symbols from display.elf...
Remote debugging using :2454
main () at display.c:15
15              value = *SW_ptr;    // read SW values
Loading section .text, size 0x1f44 lma 0x10094
Loading section .eh_frame, size 0x4 lma 0x12000
Loading section .init_array, size 0x4 lma 0x12004
Loading section .fini_array, size 0x4 lma 0x12008
Loading section .data, size 0x548 lma 0x12010
Loading section .sdata, size 0x14 lma 0x12558
Start address 0x00010094, load size 9388
Transfer rate: 82 KB/sec, 1564 bytes/write.
pc             0x10094  0x10094 <_start>
(gdb)
```

Figure 23. Starting GDB for the `display_C` example.

```
(gdb) b main
Breakpoint 1 at 0x101ac: file display.c, line 12.
(gdb) cont
Continuing.

Breakpoint 1, main () at display.c:12
12          volatile unsigned int *HEX3_HEX0_ptr = (unsigned int *) HEX3_HEX0_BASE;
(gdb) list
7       int main(void)
8       {
9           int value;
10          volatile unsigned int *SW_ptr = (unsigned int *) SW_BASE;
11          volatile unsigned int *LEDR_ptr = (unsigned int *) LEDR_BASE;
12          volatile unsigned int *HEX3_HEX0_ptr = (unsigned int *) HEX3_HEX0_BASE;
13
14          while ( 1 ){
15              value = *SW_ptr;    // read SW values
16              *LEDR_ptr = value;  // display on LEDR
(gdb)
```

Figure 24. Listing the *main* function.

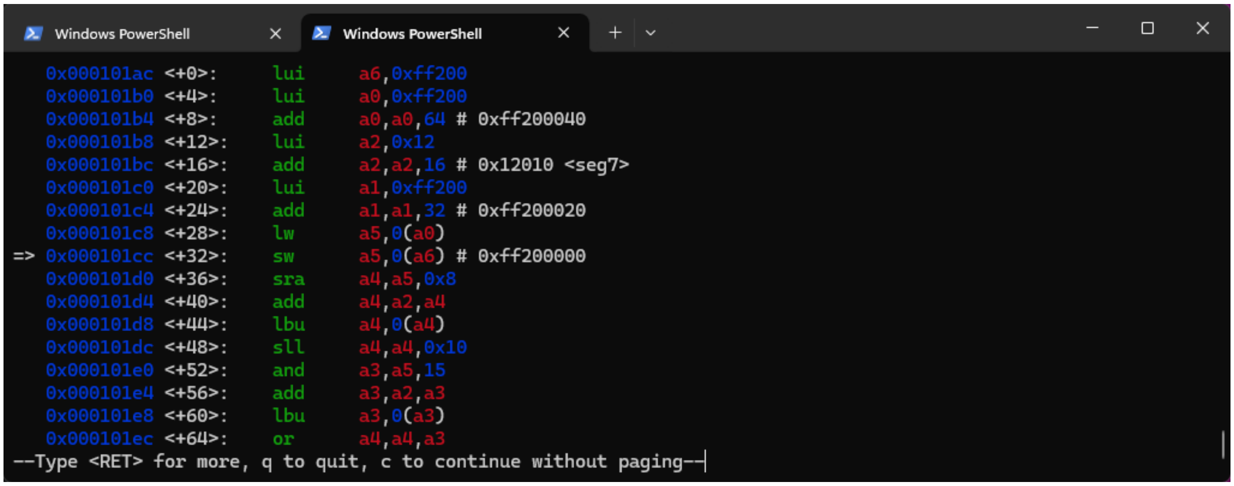
```
(gdb) b 15
Breakpoint 2 at 0x101c8: file display.c, line 15.
(gdb) cont
Continuing.

Breakpoint 2, main () at display.c:15
15              value = *SW_ptr;    // read SW values
(gdb) s
16              *LEDR_ptr = value;  // display on LEDR
(gdb) print /x value
$1 = 0x7f
(gdb)
```

Figure 25. Loading a variable from an I/O device.

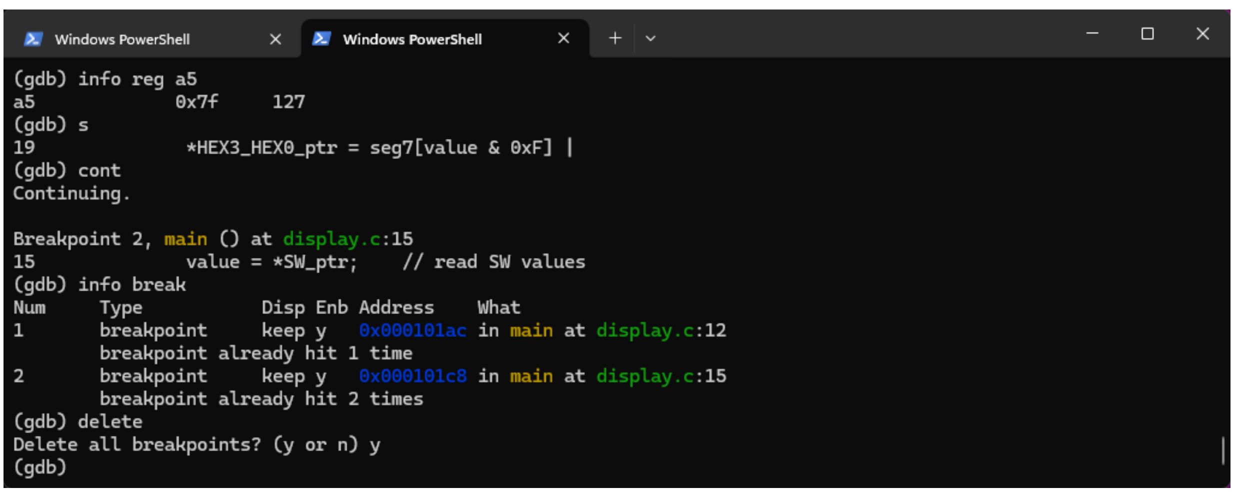
Enter the command `info break` to see the two breakpoints that are currently set, at Line 12 (*main*) and Line 15. Use the `delete` command to clear these breakpoints.

As we are finished with this example, use `detach` to disconnect from the DE1-SoC board and then `quit` from the GDB client. In the GDB Server terminal type `q` to quit.



```
   0x000101ac <+0>:     lui     a6,0xff200
   0x000101b0 <+4>:     lui     a0,0xff200
   0x000101b4 <+8>:     add     a0,a0,64 # 0xff200040
   0x000101b8 <+12>:    lui     a2,0x12
   0x000101bc <+16>:    add     a2,a2,16 # 0x12010 <seg7>
   0x000101c0 <+20>:    lui     a1,0xff200
   0x000101c4 <+24>:    add     a1,a1,32 # 0xff200020
   0x000101c8 <+28>:    lw      a5,0(a0)
=> 0x000101cc <+32>:    sw      a5,0(a6) # 0xff200000
   0x000101d0 <+36>:    sra     a4,a5,0x8
   0x000101d4 <+40>:    add     a4,a2,a4
   0x000101d8 <+44>:    lbu     a4,0(a4)
   0x000101dc <+48>:    sll     a4,a4,0x10
   0x000101e0 <+52>:    and     a3,a5,15
   0x000101e4 <+56>:    add     a3,a2,a3
   0x000101e8 <+60>:    lbu     a3,0(a3)
   0x000101ec <+64>:    or      a4,a4,a3
--Type <RET> for more, q to quit, c to continue without paging--
```

Figure 26. Disassembling C code into assembly code.



```
(gdb) info reg a5
a5             0x7f     127
(gdb) s
19                *HEX3_HEX0_ptr = seg7[value & 0xF] |
(gdb) cont
Continuing.

Breakpoint 2, main () at display.c:15
15                value = *SW_ptr;     // read SW values
(gdb) info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x000101ac in main at display.c:12
        breakpoint already hit 1 time
2       breakpoint     keep y   0x000101c8 in main at display.c:15
        breakpoint already hit 2 times
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb)
```

Figure 27. Stepping through C code.

3.2 Setting up Your Own C-Code Makefile

It is easy to customize the *Makefile* for the `display_C` example so that you can use it with any C code of your choosing. `Line 3` of the *Makefile* has to specify the name of the C-code file that has the `main` function, which designates the beginning of the program. Any `.h` header files that are used with the program can be listed in `Line 4`. Finally, any additional C source-code files can be listed in `Line 5`.

3.3 Using Interrupts with C Code

This part of the tutorial uses Nios V interrupts with C code. As in previous examples, open two PowerShell terminals. In each terminal navigate to the folder for this design example, which is `interrupt_C`. Use one terminal to start the GDB server. In the other terminal first execute `./gmake`, which builds the executable program by running the C compiler and linker, and then start the GDB Client.

Within the GDB Client enter `break main`, and then run the program using `continue`. Enter `list 37,46` to see the beginning part of the main program for this example, as displayed in Figure 28.

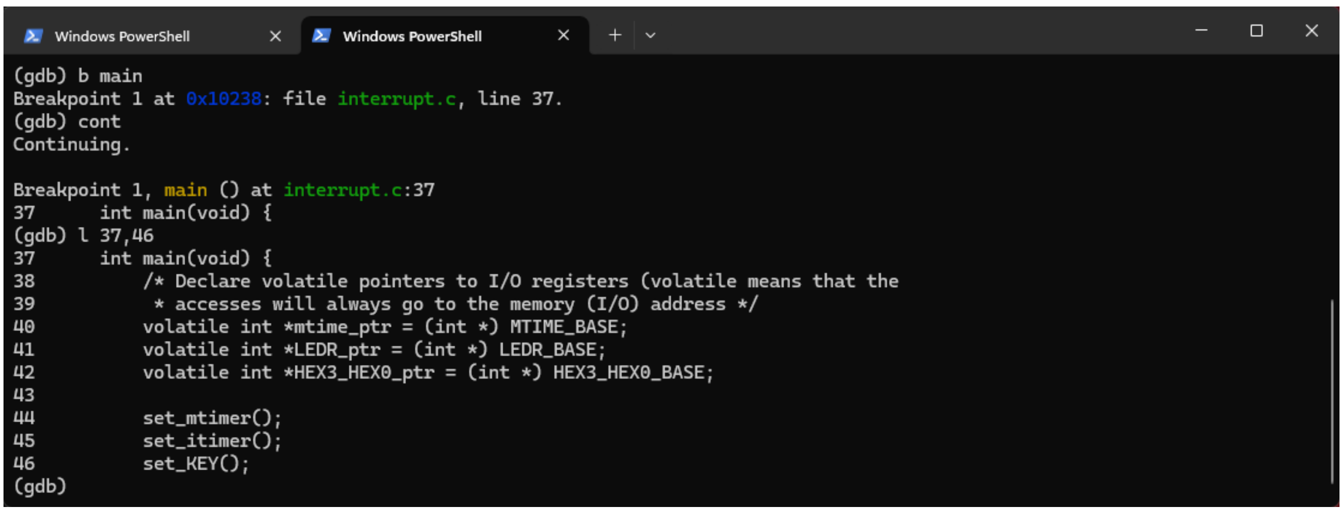


Figure 28. The *main* function.

Execute the `break 44` command and then use `continue` to run to Line 44 of the source code. Before executing the subroutine `set_mtimer()` in the program, use the command `x/4x mtime_ptr`, as depicted in Figure 29, to observe the current contents of the Nios V Machine Timer registers. These registers are referred to as *mtime* and *mtimecmp*, as described in Section 2.6. Now, execute the GDB command `next`, which causes Nios V to execute the `set_mtimer()` subroutine in the C program and then return. The `set_mtimer()` subroutine sets up the Machine Timer for one-second timeouts by reading the value of the *mtime* register, adding 100,000,000 to it, and then storing this result into *mtimecmp*. The *mtime* register then continues to increment at its 100 MHz clock rate. Rerun the command `x/4x mtime_ptr` to see the updated values of the Machine Timer registers.

Enter `break 48` and then `continue` to Line 48. Then, use `list 50,63` to show the lines of code displayed in Figure 30. The *inline assembly* code shown in the figure sets up Nios V interrupts as needed for the program. Use the command `info reg mstatus mtvec mie` to display the current values of the pertinent registers. Run the program up to Line 63 and then rerun the command `info reg mstatus mtvec mie` to see the updated register values. The *mstatus* register shows that Nios V interrupts are now enabled for machine mode, and the *mtvec*

```
(gdb) b 44
Breakpoint 2 at 0x10240: file interrupt.c, line 44.
(gdb) cont
Continuing.

Breakpoint 2, main () at interrupt.c:44
44          set_mtimer();
(gdb) x/4x mtime_ptr
0xff202100:     0xd4c0598c      0x000001d4      0xa294feff      0x000001c8
(gdb) next
45          set_itimer();
(gdb) x/4x mtime_ptr
0xff202100:     0x7d342c07      0x000001d5      0x596ed6f9      0x000001d5
(gdb) b 48
Breakpoint 3 at 0x1024c: file interrupt.c, line 48.
(gdb) c
Continuing.

Breakpoint 3, main () at interrupt.c:51
51          __asm__ volatile ("csrc mstatus, %0" :: "r"(mstatus_value));
(gdb) |
```

Figure 29. Setting the Machine Timer registers.

```
(gdb) b 48
Breakpoint 1 at 0x1024c: file interrupt.c, line 48.
(gdb) cont
Continuing.

Breakpoint 1, main () at interrupt.c:51
51          __asm__ volatile ("csrc mstatus, %0" :: "r"(mstatus_value));
(gdb) l 50,63
50          // disable interrupts
51          __asm__ volatile ("csrc mstatus, %0" :: "r"(mstatus_value));
52          mtvec_value  = (int) &handler; // set trap address
53          __asm__ volatile ("csrw mtvec, %0" :: "r"(mtvec_value));
54          // disable all interrupts that are currently enabled
55          __asm__ volatile ("csrr %0, mie" : "=r"(mie_value));
56          __asm__ volatile ("csrc mie, %0" :: "r"(mie_value));
57          mie_value = 0x50088; // KEY, itimer, mtimer, SW interrupts
58          // set interrupt enables
59          __asm__ volatile ("csrs mie, %0" :: "r"(mie_value));
60          // enable Nios V interrupts
61          __asm__ volatile ("csrs mstatus, %0" :: "r"(mstatus_value));
62
63          *(mtime_ptr + 4) = 1;   // cause a software interrupt
(gdb)
```

Figure 30. Setting up interrupts.

register contains the address of the interrupt `handler` routine. The *mie* register has bits set that enable several sources of interrupts: Nios V software interrupts, machine timer, FPGA interval timer, and *KEY* push-buttons.

Use the command `list 63, 71` to see the remainder of the *main* program. As shown in Figure 32, it contains a loop that reads two variables from memory, called *counter* and *digit*. The *counter* variable is displayed in binary on

```
(gdb) info reg mstatus mtvec mie
mstatus        0x3800   SD:0 VM:00 MXR:0 PUM:0 MPRV:0 XS:0 FS:1 MPP:3 HPP:0 SPP:0 MPIE:0 HPIE:0 SPIE:0 UPIE:0 MIE:0 HIE:
0 SIE:0 UIE:0
mtvec          0x0      0x0
mie            0x0      0
(gdb) break 63
Breakpoint 2 at 0x10278: file interrupt.c, line 63.
(gdb) cont
Continuing.

Breakpoint 2, main () at interrupt.c:63
63          *(mtime_ptr + 4) = 1;   // cause a software interrupt
(gdb) info reg mstatus mtvec mie
mstatus        0x3888   SD:0 VM:00 MXR:0 PUM:0 MPRV:0 XS:0 FS:1 MPP:3 HPP:0 SPP:0 MPIE:1 HPIE:0 SPIE:0 UPIE:0 MIE:1 HIE:
0 SIE:0 UIE:0
mtvec          0x10350  0x10350 <handler>
mie            0x50088  327816
(gdb)
```

Figure 31. Nios V control registers.

```
(gdb) l 63, 71
63          *(mtime_ptr + 4) = 1;   // cause a software interrupt
64
65          *HEX3_HEX0_ptr = 0x3f;  // show 0 on HEX0
66
67          while (1) {
68              *LEDR_ptr = counter;
69              *HEX3_HEX0_ptr = bit_codes[digit]; // display in decimal
70          }
71      }
(gdb)
```

Figure 32. The main program loop.

the *LEDR* lights, and a decimal number is displayed on *HEX0* based on the value of the *digit* variable. Both of these variables are updated by interrupt service routines that respond to hardware timers.

The interrupt *handler* for this program is given in Figure 33. It uses assembly code to read the contents of the Nios V *mcause* control register, and then uses this value to call the appropriate interrupt service routine. Assume that we wish to trace the execution of the program when an *interval timer* interrupt occurs. In the GDB Client enter the command `break itimer_ISR`. Type `continue`. When the breakpoint has been reached, enter `list`, as displayed in Figure 34. Type *step* to execute the next line of code, as depicted in Figure 35. To see the value of the *KEY_dir* variable loaded by the program enter `print KEY_dir`. Type `step` to update the `digit` variable. Enter the `step` command until the program returns from this interrupt service routine to the interrupted program.

```
/*******************************************************************
 * Trap handler: determine what caused the interrupt and call the
 *  appropriate subroutine.
 ******************************************************************/
void handler (void){
    int mcause_value;
    __asm__ volatile ("csrr %0, mcause" : "=r"(mcause_value));
    if (mcause_value == 0x80000003) // software interrupt
        SWI_ISR();
    else if (mcause_value == 0x80000007) // machine timer
        mtimer_ISR();
    else if (mcause_value == 0x80000010) // interval timer
        itimer_ISR();
    else if (mcause_value == 0x80000012) // KEY port
        KEY_ISR();
    // else, ignore the trap
}
```

Figure 33. The interrupt handler.

```
Windows PowerShell    Windows PowerShell
(gdb) break itimer_ISR
Breakpoint 4 at 0x10310: file handlers.c, line 57.
(gdb) cont
Continuing.

Breakpoint 4, itimer_ISR () at handlers.c:57
57          *timer_ptr = 0; // clear the interrupt
(gdb) l
52
53      // FPGA interval timer interrupt service routine
54      void itimer_ISR(void){
55          int new_digit;
56          volatile int * timer_ptr = (int *) TIMER_BASE;
57          *timer_ptr = 0; // clear the interrupt
58          new_digit = digit + KEY_dir;    // inc/dec the digit
59          if (new_digit < 10 && new_digit > -1)
60              digit = new_digit;  // decimal (0 to 9)
61      }
(gdb) |
```

Figure 34. A breakpoint at an interrupt service routine.

Enter the `delete` command to clear all breakpoints. Then, use `continue &` to run the program in the *background*. Observe on the DE1-SoC board that a binary counter is displayed on the *LEDR* lights, and a digit counter appears on *HEX0*. If you press any *KEY* push-button, then the direction of counting for the digit is reversed.

Stop the running program by executing the `interrupt` command. We are now finished with this example, so use `detach` to disconnect from the DE1-SoC board and then `quit` from the GDB client. Also, quit from the GDB Server in its terminal window.

```
(gdb) s
58          new_digit = digit + KEY_dir;    // inc/dec the digit
(gdb) print KEY_dir
$1 = 1
(gdb) s
59          if (new_digit < 10 && new_digit > -1)
(gdb) s
60              digit = new_digit;  // decimal (0 to 9)
(gdb) s
main () at interrupt.c:65
65          *HEX3_HEX0_ptr = 0x3f;  // show 0 on HEX0
(gdb)
```

Figure 35. Servicing an interval timer interrupt.

3.4 Using a Terminal to Print from C Code

As a final example we will show how you can use *printf* in C code from within the GDB Client. This example can be found in the `print_C` design files folder. As in previous examples, open two PowerShell terminals. In each terminal navigate to the folder for this design example. Use one terminal to start the GDB server. In the other terminal first execute `./gmake` to build the executable program, and then start the GDB Client. Now, open a *third* PowerShell terminal and navigate again to the `print_C` folder. Execute the command `./gmake TERMINAL`. This command opens a program called the *nios2-terminal*, which allows for text-based communication between the GDB Client and the DE1-SoC board via its *JTAG UART*.

Within the GDB Client enter `break main`, and then run the program using `continue`. Enter `list 4,20`, as displayed in Figure 36. The program uses an endless loop to read from the *SW* switches and *KEY* push-buttons. Whenever any KEY is pressed, the value read from the SW switches at that time is displayed as a hexadecimal value by calling the *printf* library routine. The output from *printf* appears on the *nios2-terminal* window.

```
(gdb) l 4,20
4       int main(void)
5       {
6           int SW_value, KEY_value;
7           volatile unsigned int *SW_ptr = (unsigned int *) SW_BASE;
8           volatile unsigned int *KEY_ptr = (unsigned int *) KEY_BASE;
9           volatile unsigned int *LEDR_ptr = (unsigned int *) LEDR_BASE;
10
11          while ( 1 ){
12              SW_value = *SW_ptr;     // read SW values
13              *LEDR_ptr = SW_value;   // display on LEDR
14
15              // display on Terminal
16              if ((KEY_value = *(KEY_ptr+3)) != 0) {
17                  *(KEY_ptr+3) = KEY_value;               // clear KEY port
18                  printf ("SW: 0x%0x\n", SW_value);       // print in hexadecimal
19              }
20          }
(gdb)
```

Figure 36. The *printf* example.

Use `continue &` to run the program in the background. Try different settings of the *SW* switches and press any *KEY* push-button to see the corresponding value displayed on the *nios2-terminal*.

In the GDB Client enter the `interrupt` command to stop the running program. Then, use `detach` to disconnect from the DE1-SoC board and then `quit` from the GDB client. Also, quit from the GDB Server in its terminal window. Finally, close the *nios2-terminal* by typing `^C` in its window.

Appendix A: GDB Command Reference

Examples of GDB commands are summarized below. You can often execute a command by typing only part of its name: for example **s** executes the *step* command, **b** executes *break*, and **cont** executes *continue*.

step	execute a single source-code line
`<CR>`	simply pressing the `ENTER` key (*Carriage Return*) repeats the *last command*
step &	execute a single instruction, in the *background*
step *n*	execute *n* source-code lines
stepi	execute a single Nios V machine instruction (not used in the tutorial)
continue	run the program from its current location
continue &	run the program from its current location, in the *background*
interrupt	stop the execution of a program that is running in the *background*
next	execute the next source-code line, stepping over a subroutine call
break *k*	set a breakpoint at Line k in source code
clear *k*	clear the breakpoint at Line k
info reg [*name*, ...]	show the current value of Nios V register(s) *name*, ...
info symbol address	give the address of a symbol
info address symbol	give the symbol at an address
info break	list all active breakpoints
set $name = *value*	Set the contents of Nios V register *name* to *value*
set name = *value*	Set the contents of *name* (for example, a variable) to *value*
delete	clear all active breakpoints
x *A*	display the word in memory at address *A*
x/x *A*	display the word in memory in hexadecimal at address *A*
x/4x *A*	display the four words in memory in hexadecimal starting at address *A*
x/xb *A*	display the byte in memory in hexadecimal at address *A*
print *expr*	print the value of an expression (for example, a variable)
print /x *expr*	print the value of an expression in hexadecimal
detach	disconnect the GDB Client from the target
quit	close the GDB client, or GDB Server
load	load the executable program into memory (used in Makefiles)
target remote *port*	connect to remote debugging *port* (used in Makefiles)