Debugging

Mark McDermott

Fall 2009

Agenda

- What is Debugging?
- ARM Hardware support for Debugging
- Application Program Debugging with GDB
- Xilinx Debugging
- Linux Kernel Debugging

Introduction

- Debugging is an integral part of embedded systems development
- The debugging process is defined as testing, stabilizing, localizing, and correcting errors
- Two methods of debugging
  - Hardware debugging
    - via a logic probe, logic analyzer, in-circuit emulator, or background debugger
  - Software debugging via a debugging instrument
    - A software debugging instrument is source code that is added to the program for the purpose of debugging
- Debugging types
  - Functional debugging
  - Performance debugging

Testing and Debugging

- **Testing and debugging go together like peas in a pod:**
  - Testing finds errors; debugging localizes and repairs them.
  - Together these form the "testing/debugging cycle": we test, then debug, then repeat.
  - Any debugging should be followed by a reapplication of all relevant tests, particularly regression tests. This avoids (reduces) the introduction of new bugs when debugging.
  - Testing and debugging need not be done by the same people (and often should not be).

Boult, uccs.edu

What is a Debugger?

- "A software tool that is used to detect the source of program or script errors, by performing step-by-step execution of application code and viewing the content of code variables."
- A debugger is not an IDE
  - Though the two can be integrated, they are separate entities.
- A debugger loads in a program (compiled executable, or interpreted source code) and allows the user to trace through the execution.
- Debuggers typically can do disassembly, stack traces, expression watches, and more.

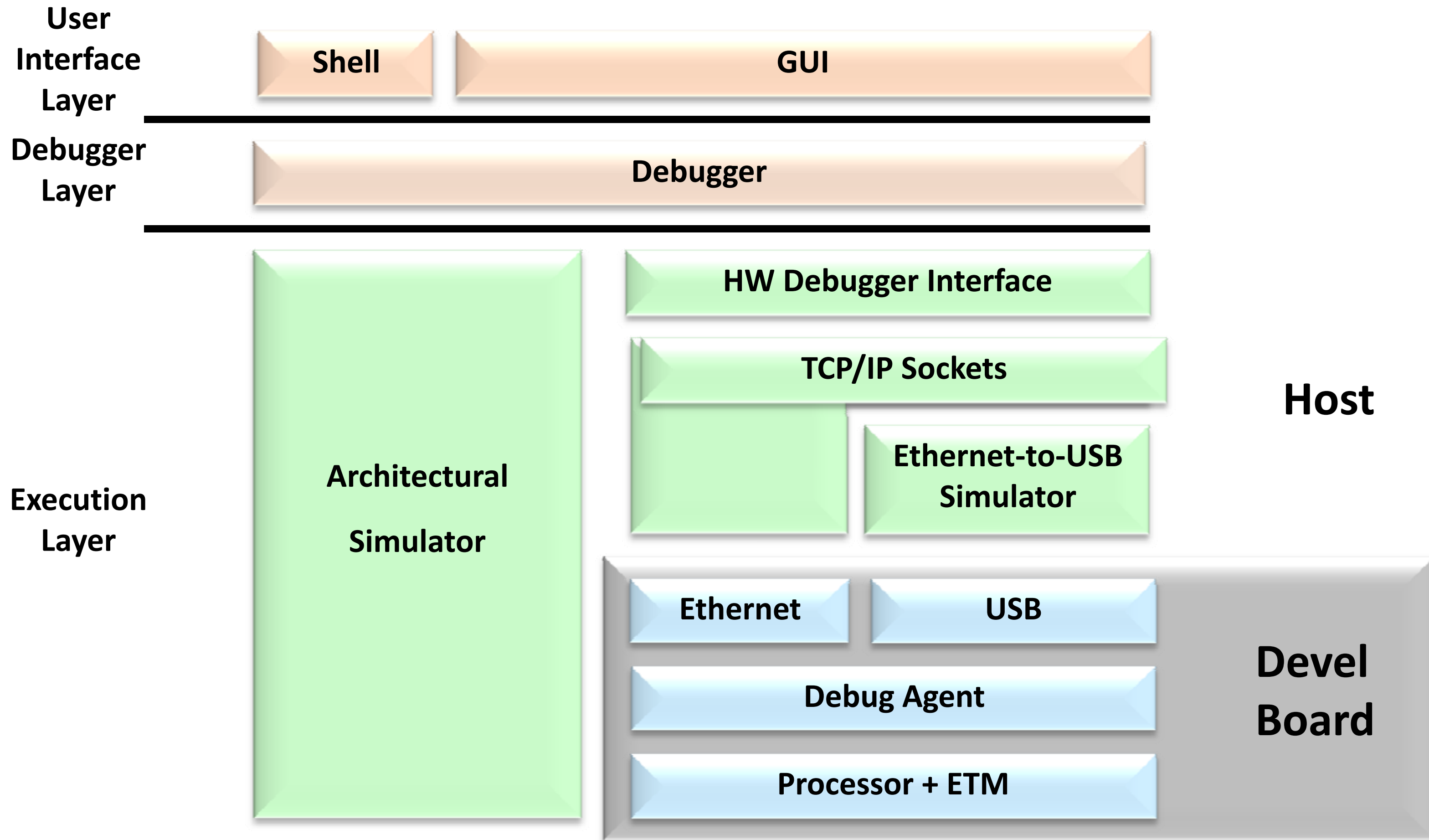
Boult, uccs.edu

Types of bugs

- Compile time: syntax, spelling, static type mismatch.
  - Usually caught with compiler
- Design: flawed algorithm.
  - Incorrect outputs
- Program logic (if/else, loop termination, select case, etc).
  - Incorrect outputs
- Memory problems: null pointers, array bounds, bad types, leaks.
  - Runtime exceptions
- Interface errors between modules, threads, programs (in particular, with shared resources: sockets, files, memory, etc).
  - Runtime Exceptions
- Off-nominal conditions: failure of some part of software of underlying machinery (network, etc).
  - Incomplete functionality
- Deadlocks: multiple processes fighting for a resource.
  - Freeze ups, never ending processes

Boult, uccs.edu

Typical Integrated Debug Environment



ARM Hardware support for Debugging

Embedded Processor Debugging

There are two main ways of debugging an embedded processor in a SOC

- Conventional Debug (invasive – core halted)
  - Set breakpoints and/or watch-points to halt the core at the specific activity
  - Use a debug connection to examine/modify register/memory and perform single step execution
- Trace (non-invasive – core runs at full speed)
  - Collect instruction execution and/or data transfers
  - Deliver off-chip in real time
  - Merge trace data with source code on development workstation for later analysis

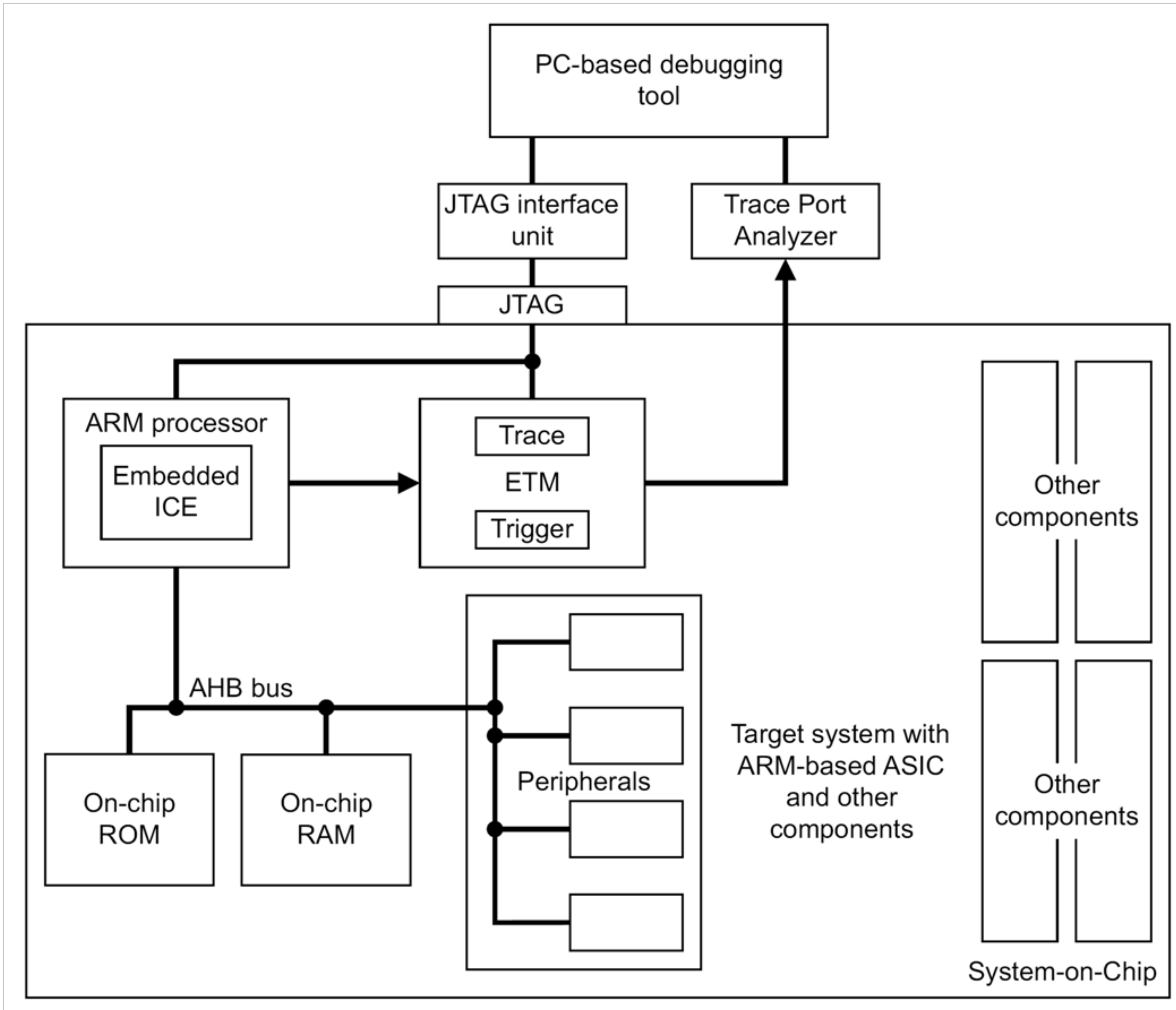
ARM Embedded Trace Macrocell (ETM)

- The ARM processor has hardware support for debugging in the form of a macrocell which can be instantiated with the processor during the design process.
- The main features of an ETM are:
- Trace generation
  - Outputs information that help you to understand the operation of the processor.
  - The trace protocol provides a real-time trace capability for processor cores that are deeply embedded in much larger ASIC designs.
    - Cannot determine how the processor core is operating by observing the pins of the ASIC, because the ASIC typically includes significant amounts of on-chip memory.
- Triggering and filtering facilities
  - An extensible specification enables you to control tracing by specifying the exact set of triggering and filtering resources required for a particular application.
  - Resources include address comparators and data value comparators, counters, and sequencers.

ETM User Interface

- A software debugger provides the user interface to the ETM.
- The debugger can configure all the ETM facilities, such as the trace port, typically using a JTAG interface. The debugger also displays the trace information that has been captured.
- The ETM compresses the trace information and either:
  - Exports it through a trace port. An external Trace Port Analyzer (TPA) captures the trace information
  - Writes it directly to an on-chip Embedded Trace Buffer (ETB). The trace is read out at low speed using the JTAG interface when the trace capture is complete
- When the trace has been captured the debugger extracts the information from the TPA or ETB and decompresses it to provide a full disassembly, with symbols, of the code that was executed.
- The debugger can also link this back to the original high-level source code, providing you with a visualization of how the code was executed on the target system.

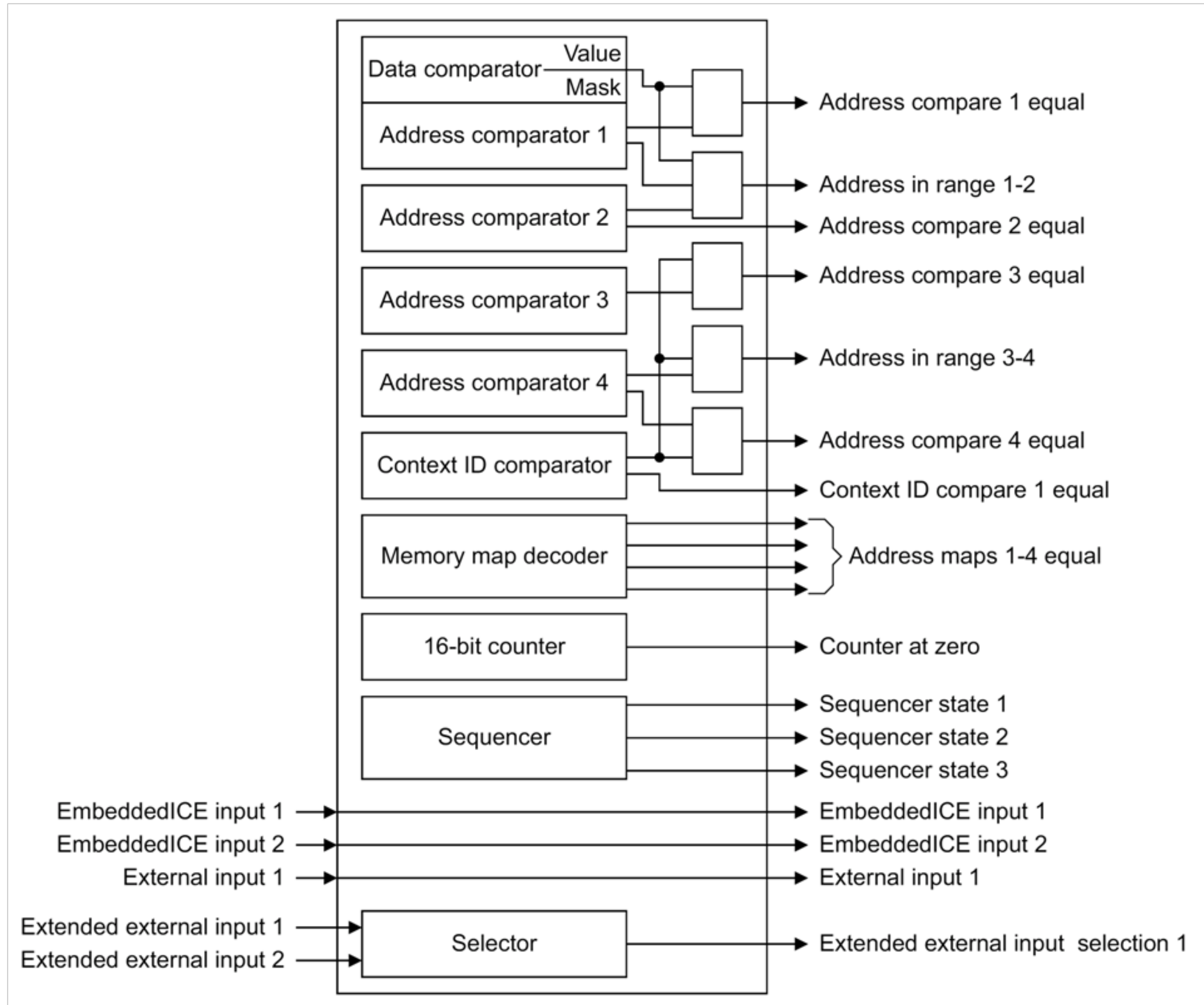
Typical ARM Debug Environment



ETM events

- **The possible ETM event resource types are:**
  - Address comparators. These can operate on both instruction and data access addresses
  - Data value comparators
  - Context ID comparators
  - Memory map decoders
  - EmbeddedICE™ module watchpoint comparators
  - Counters
  - A three-state sequencer
  - External inputs
  - Extended external inputs
  - Trace start/stop
  - Instrumentation resources, controlled by software instructions.

ETM Block Diagram



Memory Access Event Resources

- **There are five types of memory access resource:**
  - Single address comparators, used with or without data value comparators
  - Address range comparators, used with or without data value comparators
  - Context ID comparators
  - EmbeddedICE™ module watchpoint comparators
  - Device-specific memory map decoders.

Address Comparators

- **Address comparators compare either the instruction address or the data address against a user-programmed value.**
  - There are between zero and 16 single address comparators
  - Each pair can have an associated bit-masked data value comparator
- **Each comparator has several configuration bits to determine the match conditions. The available options are:**
  - instruction fetch
  - instruction execute (irrespective of condition code passed or failed)
  - instruction executed and condition code test passed
  - instruction executed and condition code test failed
  - data load or store
  - data load only
  - data store only.

GNU Debugger

GDB, the GNU Debugger

- Text-based, invoked with:

    gdb [<program_file> [<corefile>|<pid>]]

- Argument descriptions:

<program_file>	executable program file
<corefile>	core dump of program (crash records)
<pid>	process id of already running program

- Compile <program_file> with –g for debug info

    cc -g <program_file>

andrzej@csun.edu

GDB, the GNU Debugger (cont)

- **corefiles very useful in post mortem analysis (figuring out what went wrong; why did it crash?)**

- **Starting with pid very useful for programs that misbehave non-deterministically (scenario difficult to repeat; problem exhibited sporadically; field troubleshooting)**

andrzej@csun.edu

GDB - Freeware GNU Debugger

Starting GDB: gdb

Loading the symbol table:

file *my_prog*

***/my_prog* is executable file name!/**

Start GDB and load the symbol table in one step:

gdb *my_prog*

Exit GDB: quit

Executing shell commands: shell *command args*
make* is a special case: make *args

GDB – Commands

Command names may be truncated if the abbreviation is unambiguous: `s` (for step), `r` (for run)

UNIX style TAB completion for the command names.

Alternative way: complete chars.

Ex.: complete h results in:

```
handle
hbreak
help
```

Getting help:

- `help` (or h) – lists all classes of commands.
- `h command` – displays a short description of the command

GDB – Running Programs

Running a program:

`run` (or `r`) -- creates an inferior process that runs your program.

- **if there are no execution errors the program will finish and results will be displayed**
- **in case of error, the GDB will show:**
  - the line the program has stopped on and
  - a short description of what it believes has caused the error

There is a certain information that affects the execution of a program:

- **program's arguments**
- **program's environment**
- **program's working directory**
- **the standard input and output**

GDB – Program's arguments

Specifying arguments for your program:

As arguments to `run`: `run` *`arg1`* *`arg2`* …

With `set args` command: `set args` *`arg1`* *`arg2`* …

- ! `run` **without arguments uses the same arguments used by the previous** `run`**.**
- ! `set args` **without arguments – removes all arguments.**
- ! `show args` **command shows the arguments your program has been started with.**

GDB – Program's environment

Changing the PATH environment variable:

`path` *`dir`* – add the directory *`dir`* at the beginning of the PATH variable. You may specify several directory names separated by ':' or white space.

`show paths` – displays the search paths for executables.

Changing the working directory:

`cd` *`dir`* – to change the working directory

Redirecting output:

`run >` *`outfile`* direct the output to the file *`outfile`*.

GDB - Debugging an already-running process

From inside GDB:

`attach` *`process-id`*

/ To get the process ID use the UNIX command `ps` /

From outside GDB:

`gdb` *`my_prog process-id`*

The first thing GDB does after arranging to debug the specified process is to stop it.

`detach` – detaches the currently attached process from the GDB control. A detached process continues its own execution.

GDB – Breakpoints and watchpoints

Breakpoints and watchpoints allow you to specify the places or the conditions where you want your program to stop.

`break` *`arg`* – stops when the execution reaches the specified line

`/` *`arg`* – function name, line number, +/- offset /

`watch` *`expr`* – stops whenever the value of the expression changes

`clear [`*`arg`*`]` - Without arguments deletes any breakpoint at the next instruction to be executed in the current stack frame

`delete [`*`bnum`*`]`- Without arguments deletes all breakpoints.

GDB – Examining variables

Global variables can be examined from every point in the source file.

Local variables – can be examined only in their scope or using:

`file::variable` or `function::variable`

The variable type: `ptype var`

Current value: `print var`

Automatic display: `display var` – adds `var` to the *automatic display list.*

`undisplay dnum`

Specifying the output format (`x, o, d, u, t, a, f,` and `c`) **:**

`print /t var` - prints the value of `var` in binary format

GDB – Value history

The *value history* keeps the values printed by the `print` command.

Previously printed values can be accessed by typing `$` followed by their history number.

`$` - refers to the most recent value and

`$$n` - refers to the *n-th* value from the end.

`show values [n/+]`

Without argument – the last 10 values.

n – 10 values centered around *n*

+ – 10 values after the last printed

GDB – Stepping through the program

`step [count]` – program execution continue to next source line going into function calls.

`next [count]` – program execution continue to the next source line omitting function calls.

`continue` – resume program execution

`until` – continue until the next source line in the current stack frame is reached. /useful to exit from loops/

GDB – Altering execution

Returning from a function

`finish` – forced return

`return [ret_value]` – pops the current stack frame

Continuing at different address

`jump line_num|*address`

Altering the value of a variable

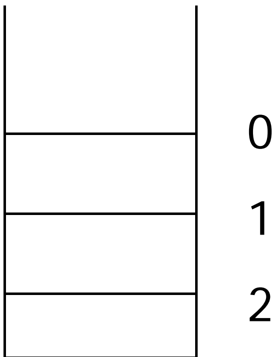
`set i=256`

Proceeding to a specified point:

`until [line_num|*address |function_name]`

GDB – The stack frame

Stack frames are identified by their addresses, which are kept in the *frame pointer* register.



Selecting a frame:

frame *n*|*addr*

up *n*

down *n*

Information about the current frame

frame – brief description

info args – shows function arguments

info locals – shows local variables

GDB – Convenience variables

Convenience variables are used to store values that you may want to refer later. Any string preceded by `$` is regarded as a convenience variable.

Ex.: `$table = *table_ptr`

There are several automatically created convenience variables:

- `$pc` – program counter
- `$sp` – stack pointer
- `$fp` – frame pointer
- `$ps` – processor status
- `$_` – contains the last examined address
- `$__` – the value in the last examined address
- `$_exitcode` – the exit code of the debugged program

GDB – Examining memory

The `x` command (for "examine"):

`x/nfu addr` – specify the number of units (*n*), the display format (*f*) and the unit size (*u*) of the memory you want to examine, starting from the address *addr*. Unit size can be – `b`, `h` (half), `w` and `g` (giant).

`x addr` – start printing from the address *addr*, others default

`x` – all default

Registers

Registers names are different for each machine. Use `info registers` to see the names used on your machine.

GDB has four "standard" registers names that are available on most machines: program counter, stack pointer, frame pointer and processor status.

GDB – Additional process information

`info proc` – summarize available information about the current process.

`info proc mappings` – address range accessible in the program.

`info proc times` – starting time, user CPU time and system CPU time for your program and its children.

`help info !`

`info signals` – information about the system signals and how GDB handles them.

DDD - The Data Display Debugger

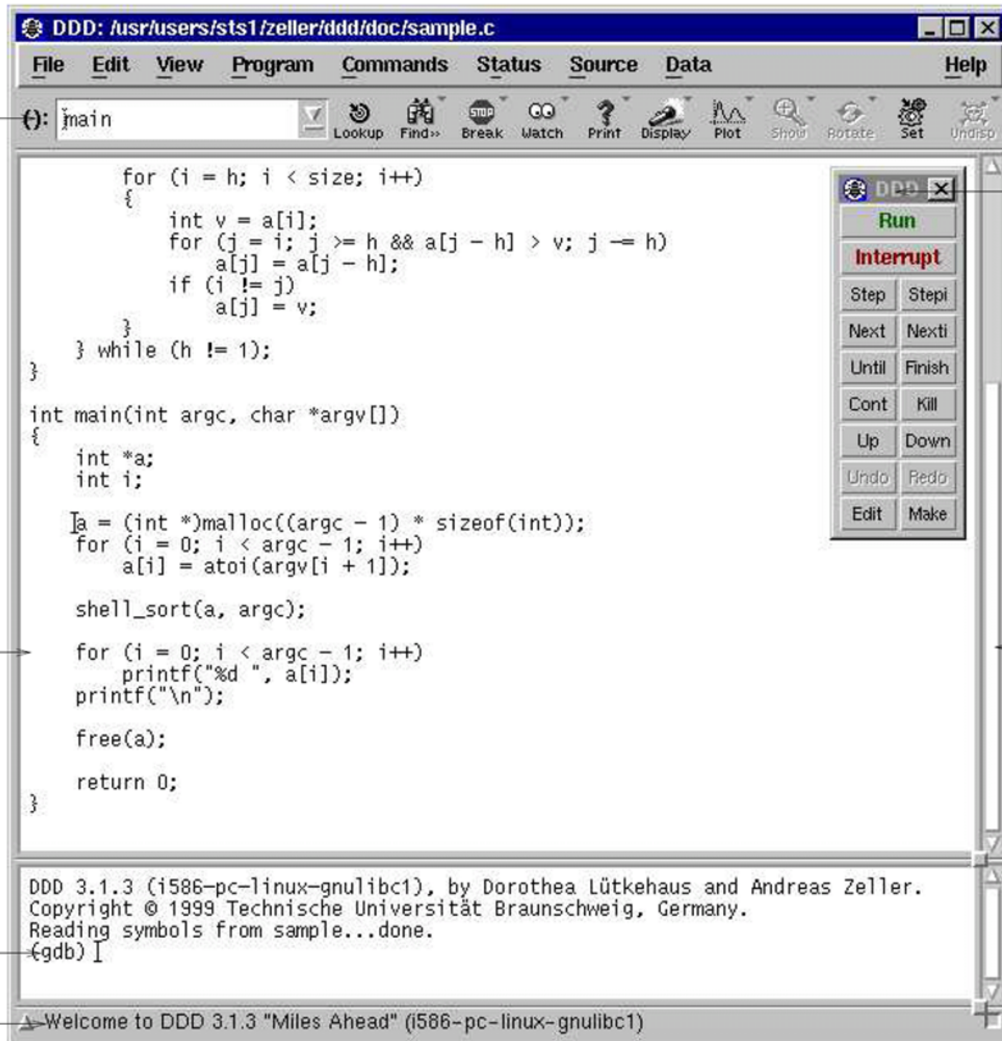
DDD is a GUI debugger that can work with several inferior debuggers including GDB.

GDB commands can be typed in the debugger console window.

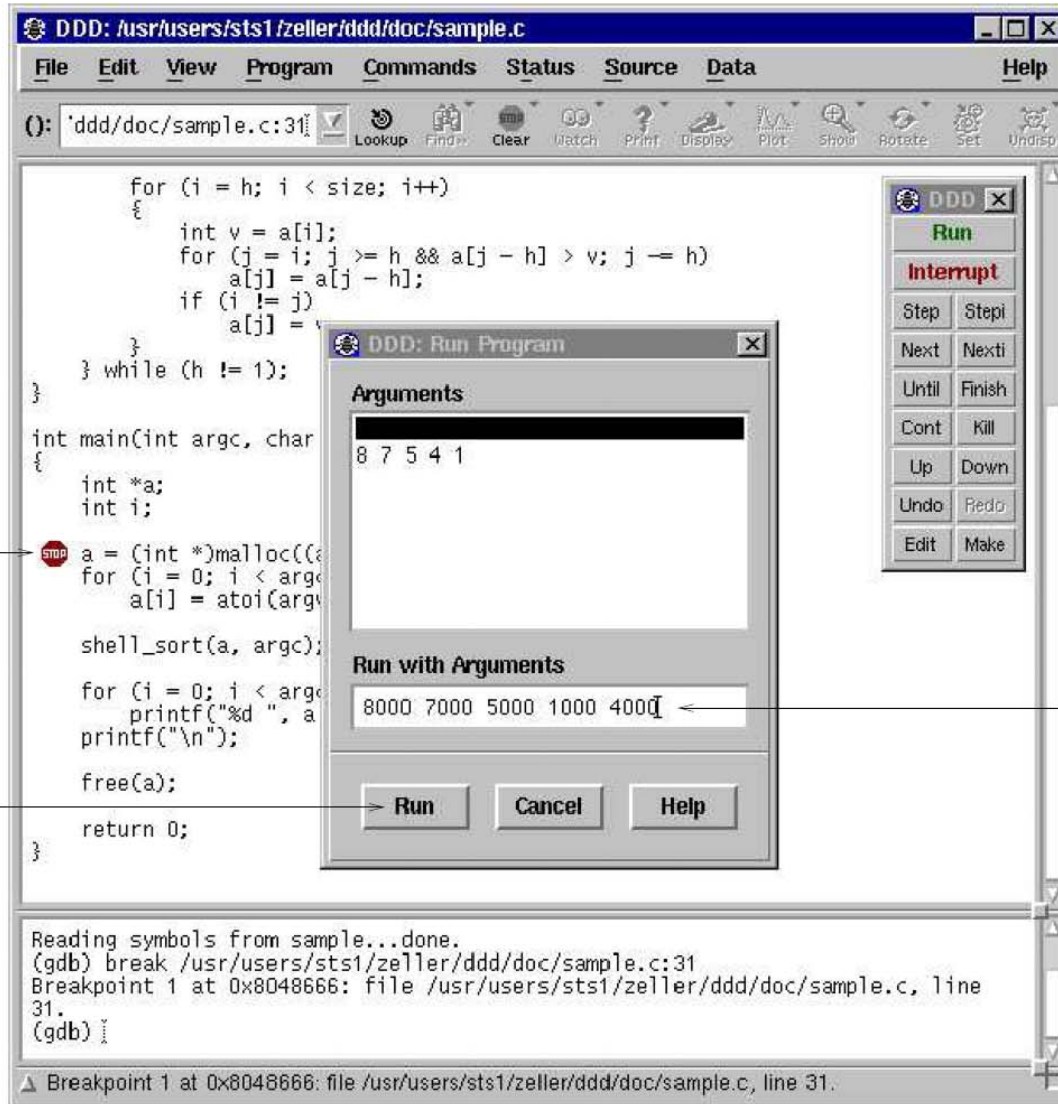
DDD sets some GDB settings automatically and will not work correctly if you change them. These are:

```
set height 0
set width 0
set verbose off
set prompt (gdb) !
```

Initial startup window



Setting breakpoints



Breakpoint

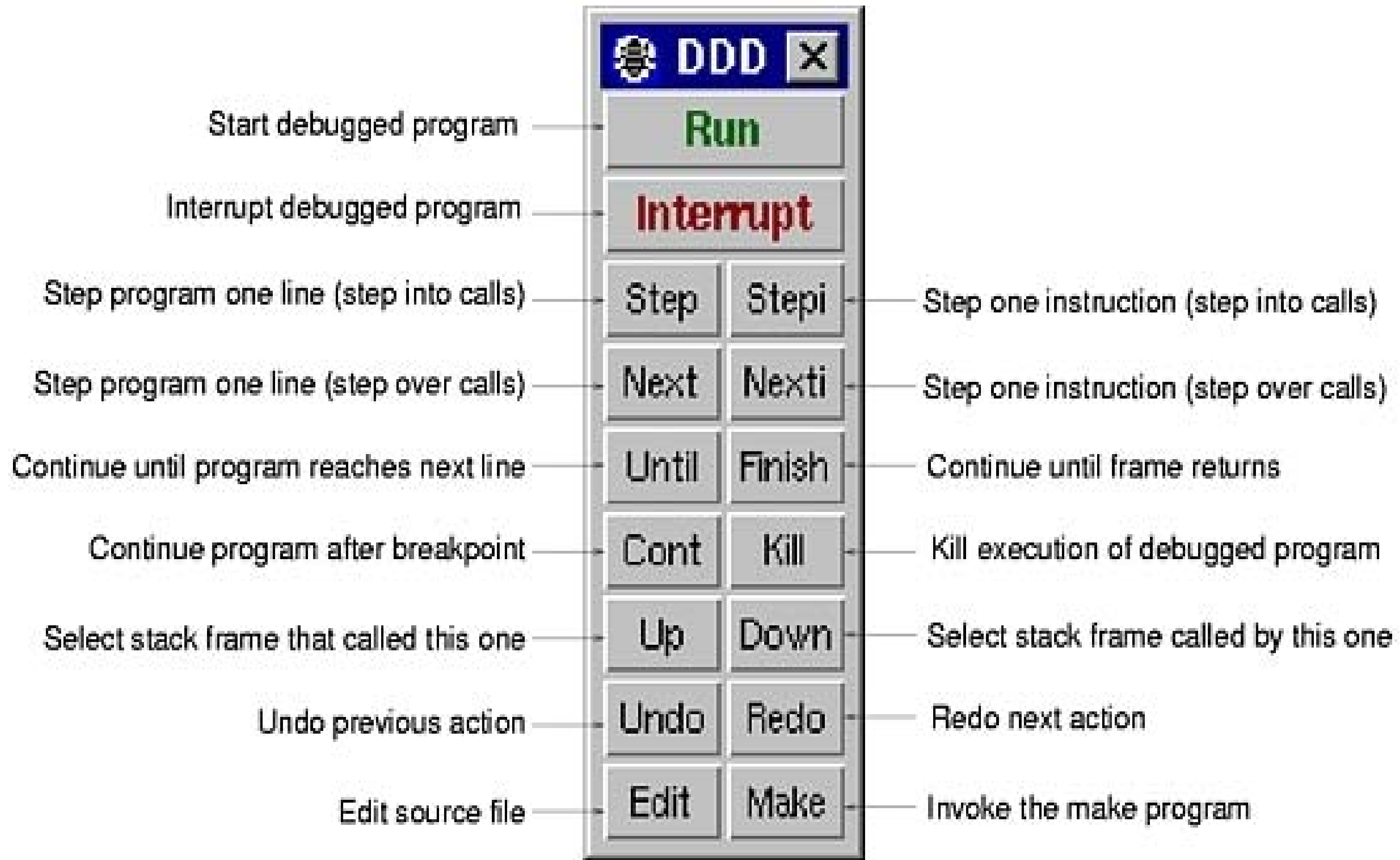
Arguments

Click here to run

DDD – GUI Advantages

- **Frequently used commands are on the toolbars, have assigned shortcut keys or can be done just with a mouse click.**
- **Easy browsing through the source !**
- **Examining current variables values directly – by placing the mouse pointer over them.**
- **Possibility to graphically display the program data.**
- **Help menu – What now? – very helpful !**

DDD – Command menu



The Command Tool

gdb - Example

```
(gdb) list 1                    ---> can use "l" for "list"
1       /* REVERSE.C */
2
3       #include <stdio.h>
4
5       /* Function Prototype */
6       void reverse ();
7
8       /****************************************************************/
9
10       main ()
(gdb) l                                   ---> same as "list"; continues from the previous
11
12        {
13 char str [100]; /* Buffer to hold reversed string */
14
15 reverse ("cat", str); /* Reverse the string "cat" */
16 printf ("reverse (\"cat\") = %s\n", str); /*       Display */
17 reverse ("noon", str); /* Reverse the string "noon" */
18 printf ("reverse (\"noon\") = %s\n", str); /* Display */
19       }
20
(gdb) _
```

andrzej@csun.edu

gdb - Example

```
(gdb) break main                    ---> set a breakpoint in function main
Breakpoint 1 at 0x29f4: file reversefirst.c, line 15.
(gdb) break 16                      ---> set a breakpoint in line 16 (current source)
Breakpoint 2 at 0x2a0c: file reversefirst.c, line 16.
(gdb) break reverse                 ---> set a breakpoint in function reverse
Breakpoint 3 at 0x2a80: file reversefirst.c, line 33.
(gdb) info break                    ---> display information on breakpoints
Num Type        Disp Enb Address    What
1   breakpoint  keep y   0x000029f4 in main at reversefirst.c:15
2   breakpoint  keep y   0x00002a0c in main at reversefirst.c:16
3   breakpoint  keep y   0x00002a80 in reverse at reversefirst.c:33
(gdb) run
Starting program: /home/usersNN/userID/reverse/reverse

Breakpoint 1, main () at reversefirst.c:15
15 reverse ("cat", str); /* Reverse the string "cat" */
(gdb) ontinue                       ---> you can use "c" as well

Breakpoint 3, reverse (before=0x40001028 "cat", after=0x7f7f0958 "")
   at reversefirst.c:33
33 len = strlen (before);
(gdb) _
```

andrzej@csun.edu

gdb - Example

```
(gdb) backtrace                                   ---> show the execution stack
#0  reverse (before=0x40001028 "cat", after=0x7f7f0958 "") at reversefirst.c:33
#1  0x2a0c in main () at reversefirst.c:15
(gdb) l
28 {
29          int i;
30 int j;
31 int len;
32
33 len = strlen (before);
34
35 for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
36          after[i] = before[j];
37
(gdb) next                                ---> execute next line
35 for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
(gdb) n                                           ---> same as “next”
36          after[i] = before[j];
(gdb) _
```

gdb - Example

```
(gdb) print after[i]                  ---> display data (expression)
$1 = 0 '\000'
(gdb) p before[j]                     ---> same as "print"
$2 = 116 't'
(gdb) _
(gdb) n
35   for (j = len - 1, i = 0; j >= 0; j--, i++) /* Reverse loop */
(gdb) p after                         ---> print
$4 = 0x7f7f0958 "t"
(gdb) p before
$5 = 0x40001028 "cat"
(gdb) c
Continuing.
Breakpoint 2, main () at reversefirst.c:16
16   printf ("reverse (\"cat\") = %s\n", str); /*        Display */
17   (gdb) _
```

andrzej@csun.edu

gdb - Example

```
(gdb) n
reverse ("cat") = tac
17 reverse ("noon", str); /* Reverse the string "noon" */
(gdb) s
Breakpoint 3, reverse (before=0x40001030 "noon", after=0x7f7f0958 "tac")
   at reversefirst.c:33
33 len = strlen (before);
(gdb) return 0
Make reverse return now? (y or n) y
#0  main () at reversefirst.c:18
18 printf ("reverse (\"noon\") = %s\n", str); /* Display */
(gdb) p str
$4 = "tac", '\000' <repeats 96 times>
(gdb) n
reverse ("noon") = tac    ---> this is the output from the program
19     }
(gdb) quit
$ _
```

andrzej@csun.edu

Remote debugging

- gdbserver is a control program for Unix-like systems, which allows you to connect your program with a remote GDB via target remote---but without linking in the usual debugging stub.
- GDB and gdbserver communicate via either a serial line or a TCP connection, using the standard GDB remote serial protocol.
- *On the target machine,* you need to have a copy of the program you want to debug.
  - gdbserver does not need your program's symbol table, so you can strip the program if necessary to save space.
  - GDB on the host system does all the symbol handling.
  - To use the server, you must tell it how to communicate with GDB; the name of your program; and the arguments for your program. The usual syntax is:

```
target> gdbserver comm program [ args ... ]
```

comm is either a device name (to use a serial line) or a TCP hostname and portnumber.

gdbserver waits passively for the host GDB to communicate with it.

  - For example to use a TCP connection instead of a serial line:

```
target> gdbserver host:2345 emacs foo.txt
```

Linux Kernel Debugging

Overview

- **Several tools are available**
- **Some are more difficult to set up and learn**
- **All of them have inherent limitations**
  - Intrusive, may change the kernel behavior
  - Kernel might crash the tool

Baker/Wang, fsu.edu

Debugging Support in the Linux Kernel

- Under the "kernel hacking" menu in the "menuconfig" setup
  - Not supported by all architectures
- `CONFIG_DEBUG_KERNEL`
  - Enables other debugging features
- `CONFIG_DEBUG_SLAB`
  - Checks kernel memory allocation functions
    - Memory overrun
    - Missing initialization
- `CONFIG_DEBUG_PAGEALLOC`
  - Pages are removed from the kernel address space when freed
- `CONFIG_DEBUG_SPINLOCK`
  - Catches operations on uninitialized spinlocks and double unlocking
- `CONFIG_DEBUG_SPINLOCK_SLEEP`
  - Checks for attempts to sleep while holding a spinlock
- `CONFIG_INIT_DEBUG`
  - Checks for attempts to access initialization-time memory
- `CONFIG_DEBUG_INFO`
  - Enables `gdb` debugging

Baker/Wang, fsu.edu

Debugging Support in the Kernel

- `CONFIG_MAGIC_SYSRQ`
  - For debugging system hangs
- `CONFIG_DEBUG_STACKOVERFLOW`
  - Helps track down kernel stack overflows
- `CONFIG_DEBUG_STACK_USAGE`
  - Monitors stack usage and makes statistics available via magic SysRq key
- `CONFIG_KALLSYMS`
  - Causes kernel symbol information to be built into the kernel
- `CONFIG_IKCONFIG`
- `CONFIG_IKCONFIG_PROC`
  - Causes kernel configurations to be made available via `/proc`
- `CONFIG_ACPI_DEBUG`
  - Enables debugging for power management
- `CONFIG_DEBUG_DRIVER`
- `CONFIG_SCSI_CONSTANTS`
- `CONFIG_PROFILING`
  - For performance tuning

Baker/Wang, fsu.edu

`printk` (vs. `printf`)

- Lets one classify messages according to their priority by associating with different loglevels
```
printk(KERN_DEBUG "Here I am: %s:%i\n",
  __FILE__, __LINE__);
```
- Eight possible loglevels (0 - 7), defined in `<linux/kernel.h>`

Baker/Wang, fsu.edu

printk (vs. printf)

- **KERN_EMERG**
  - For emergency messages
- **KERN_ALERT**
  - For a situation requiring immediate action
- **KERN_CRIT**
  - Critical conditions, related to serious hardware or software failures
- **KERN_ERR**
  - Used to report error conditions; device drivers often use it to report hardware difficulties
- **KERN_WARNING**
  - Warnings for less serious problems
- **KERN_NOTICE**
  - Normal situations worthy of note (e.g., security-related)
- **KERN_INFO**
  - Informational messages
- **KERN_DEBUG**
  - Used for debugging messages

Baker/Wang, fsu.edu

printk (vs. printf)

- **Without specified priority**
  - DEFAULT_MESSAGE_LOGLEVEL = KERNEL_WARNING
- **If current priority < console_loglevel**
  - console_loglevel initialized to DEFAULT_CONSOLE_LOGLEVEL
  - Message is printed to the console one line at a time

Baker/Wang, fsu.edu

`printk` (vs. `printf`)

- **If both `klogd` and `syslogd` are running**
  - **Messages are appended to `/var/log/messages`**
- **`klog` daemon doesn't save consecutive identical lines, only the first line + the number of repetitions**

Baker/Wang, fsu.edu

`printk` (vs. `printf`)

- `console_loglevel` can be modified using `/proc/sys/kernel/printk`
  - Contains 4 values
    - Current loglevel
    - Default log level
    - Minimum allowed loglevel
    - Boot-timed default loglevel
  - `echo 6 > /proc/sys/kernel/printk`

Baker/Wang, fsu.edu

How Messages Get Logged

- `printk` writes messages into a circular buffer that is `__LOG_BUF_LEN` bytes
  - Wakes any process that is waiting for messages
  - If the buffer fills up, `printk` wraps around and overwrite the beginning of the buffer
  - Can specify the `-f <file>` option to `klogd` to save messages to a specific file

Baker/Wang, fsu.edu

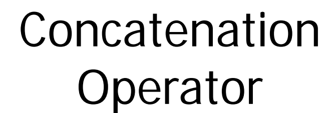
How Messages Get Logged

- Reading from `/proc/kmsg` consumes data
- `syslog` system call can leave data for other processes (try `dmesg` command)

Turning the Messages On and Off at Compile Time

- **Need to recompile to turn messages on and off**
  - Faster than using C conditionals
- **In a header file (e.g., `scull.h`)**

```
#undef PDEBUG /* undef it, just in case */
#ifdef SCULL_DEBUG
#  ifdef __KERNEL__
#    define PDEBUG(fmt, args...) \
        printk(KERN_DEBUG “scull:  “ fmt, ## args)
#  endif
#endif
```



Baker/Wang, fsu.edu

Turning the Messages On and Off

- **In the `Makefile` file**

```
DEBUG = y

ifeq ($(DEBUG),y)
  DEBFLAGS = -O -g -DSCULL_DEBUG
  # "-O" is needed to expand inlines
else
  DEBFLAGS = -O2
endif

CFLAGS += $(DEBFLAGS)
```

Baker/Wang, fsu.edu

Rate Limiting

- Too many messages may overwhelm the console
- To reduce repeated messages, use
  - `int printk_ratelimit(void);`
- Example

```
if (printk_ratelimit()) {
  printk(KERN_NOTICE "The printer is still on fire\n");
}
```

Baker/Wang, fsu.edu

Rate Limiting

- **To modify the behavior of `printk_ratelimit`**
  - `/proc/sys/kernel/printk_ratelimit`
    - **Number of seconds before re-enabling messages**
  - `/proc/sys/kernel/printk_ratelimit_burst`
    - **Number of messages accepted before rate limiting**

Printing Device Numbers

- Utility functions (defined as macros)
  - `int print_dev_t(char *buffer, dev_t dev);`
    - Returns the number of characters printed
  - `char *format_dev_t(char *buffer, dev_t dev);`
    - Returns `buffer`
  - `buffer` should be at least 20 bytes

Baker/Wang, fsu.edu

Debugging by Querying

- **Disadvantages of `printk`**
  - Every line causes a disk operation
    - Can be solved by prefixing the log file specified in `/etc/syslogd.conf` with "-"
    - Example (in `/etc/syslogd.conf`)
      - `mail.*  -/var/log/maillog`
- **One alternative: query the system**
  - `ps`, `netstat`, `vmstat`, `/proc`, `ioctl`, `sysfs`

Baker/Wang, fsu.edu

Using the `/proc` Filesystem

- **Exports kernel information**
- **Each file under `/proc` tied to a kernel function**

Baker/Wang, fsu.edu

Implementing Files in /proc

- `#include <linux/proc_fs.h>`
- For a read-only file, your driver must implement the following function

```
int (*read_proc) (char *page, char **start,
                  off_t offset, int count, int *eof,
                  void *data);
```

  – page points to a buffer with PAGE_SIZE bytes

Baker/Wang, fsu.edu

Implementing Files in `/proc`

- `start` points to where to store the read data
  - `start == NULL` indicates the `offset` is 0
- `eof` points to an integer, which signals that it has no more data to return
- `data` is device-specific, for internal bookkeeping

Baker/Wang, fsu.edu

Implementing Files in /proc

```
int scull_read_procmem(char *buf, char **start, off_t offset,
                          int count, int *eof, void *data) {
  int i, j, len = 0;
  int limit = count - 80; /* Don't print more than this */

  for (i = 0; i < scull_nr_devs && len <= limit; i++) {
    struct scull_dev *d = &scull_devices[i];
    struct scull_qset *qs = d->data;

    if (down_interruptible(&d->sem)) {
      return -ERESTARTSYS;
    }
    len +=
      sprintf(buf+len,"\nDevice %i: qset %i, q %i, sz %li\n",
              i, d->qset, d->quantum, d->size);
/* scan the list */
    for (; qs && len <= limit; qs = qs->next) {
      len += sprintf(buf + len, " item at %p, qset at %p\n",
                     qs, qs->data);
      if (qs->data && !qs->next) /* dump only the last item */
      for (j = 0; j < d->qset; j++) {
        if (qs->data[j]) {
          len += sprintf(buf + len, " % 4i: %8p\n", j,
                         qs->data[j]);
        }
      }
    }
    up(&scull_devices[i].sem);
  }
  *eof = 1;
  return len;
}
```

Baker/Wang, fsu.edu

Creating Your `/proc` File

- **To connect the `read_proc` function to `/proc`, call the following**

```
struct proc_dir_entry *create_proc_read_entry(
  const char *name, mode_t mode,
  struct proc_dir_entry *base, read_proc_t *read_proc,
  void *data);
```

  - `name`: name of the `/proc` file
    - Note: kernel does not check duplicate names
  - `mode`: protection mask (0 for default)

Baker/Wang, fsu.edu

Creating Your /proc File

- base: directory (NULL for /proc root)
- read_proc: read function defined to access a kernel data structure
- data: ignored by the kernel, but passed to read_proc

- To make scull_read_procmem available, call the following

```
create_proc_read_entry("scullmem", 0, NULL,
  scull_read_procmem, NULL);
```

Baker/Wang, fsu.edu

Creating Your /proc File

- **To remove `scull_read_procmem`, call the following**

```
remove_proc_entry("scullmem", NULL /* parent
  directory */);
```

  - **Note: `/proc` has no reference count**
    - **Make sure that no one is reading the file**

Baker/Wang, fsu.edu

The `ioctl` Method

- **Implement additional commands to return debugging information**
  - **Advantages**
    - **More efficient**
    - **Does not need to split data into pages**
    - **Can be left in the driver unnoticed**

Baker/Wang, fsu.edu

The `ioctl` Method

- **Implement additional commands to return debugging information**
  - Disadvantages
    - Needs to implement a user-level debugging program that is in sync with the kernel module
    - Slightly bigger module

Baker/Wang, fsu.edu

Debugging by Watching

- `strace` command
  - Shows system calls, arguments, and return values
  - No need to compile a program with the `–g` option
  - `-t` to display when each call is executed
  - `-T` to display the time spent in the call
  - `-e` to limit the types of calls
  - `-o` to redirect the output to a file

Baker/Wang, fsu.edu

Debugging by Watching

- `strace` example 1
  - `strace ls /dev > /dev/scull0`

```
[...]
open("/dev", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=24576, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
getdents64(3, /* 141 entries */, 4096) = 4088
[...]
getdents64(3, /* 0 entries */, 4096) = 0
close(3) = 0
[...]
```

Baker/Wang, fsu.edu

Debugging by Watching

- **strace example 1**

```
[...]
fstat64(1, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
write(1, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 4096) = 4000
write(1, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 96) = 96
write(1, "b\nptyxc\nptyxd\nptyxe\nptyxf\nptyy0\n"..., 4096) = 3904
write(1, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 192) = 192
write(1, "\nvcs47\nvcs48\nvcs49\nvcs5\nvcs50\nvc"..., 673) = 673
close(1) = 0
exit_group(0) = ?
```

The size of a **`scull`** quantum is 4,000 bytes

Baker/Wang, fsu.edu

Debugging by Watching

- **strace example 2**

```
strace wc /dev/scull0
[...]
open("/dev/scull0", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
read(3, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 16384) = 4000
read(3, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 16384) = 4000
read(3, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 16384) = 8
read(3, "", 16384) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}
write(1, "8865 /dev/scull0\n", 17) = 17
close(3) = 0
exit_group(0) = ?
```

wc bypasses the standard library and reads 16KB at a time

Baker/Wang, fsu.edu

Debugging System Faults

- A fault usually ends the current process, while the system continues to work
- Potential side effects
  - Hardware left in an unusable state
  - Kernel resources in an inconsistent state
  - Corrupted memory
- Common remedy
  - Reboot

Baker/Wang, fsu.edu

Oops Messages

- Dereferencing invalid pointers often results in oops messages

```
ssize_t
faulty_write(struct file *filp, const char __user *buf,
             size_t count, loff_t *pos) {
  /* make a simple fault by dereferencing a NULL pointer */
  *(int *)0 = 0;
  return 0;
}
```

Baker/Wang, fsu.edu

Oops Messages

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU: 0
EIP: 0060:[<d083a064>] Not tainted
EFLAGS: 00010246 (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000 ebx: 00000000 ecx: 00000000 edx: 00000000
esi: cf8b2460 edi: cf8b2480 ebp: 00000005 esp: c31c5f74
ds: 007b es: 007b ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
[<c0150558>] vfs_write+0xb8/0x130
[<c0150682>] sys_write+0x42/0x70
[<c0103f8f>] syscall_call+0x7/0xb
Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

Error message

Function name, 4 bytes into the function, kernel module name

Kernel address space if >= 0xc000000

Call stack

Baker/Wang, fsu.edu

Oops Messages

- **Buffer overflow**

```
ssize_t faulty_read(struct file *filp, char __user *buf,
                    size_t count, loff_t *pos) {
  int ret; char stack_buf[4];

  memset(stack_buf, 0xff, 20); /* buffer overflow */
  if (count > 4) {
    count = 4; /* copy 4 bytes to the user */
  }
  ret = copy_to_user(buf, stack_buf, count);
  if (!ret) { return count; }
  return ret;
}
```

Baker/Wang, fsu.edu

Oops Messages

```
EIP: 0010:[<00000000>]
Unable to handle kernel paging request at virtual address ffffffff
printing eip:
ffffffff
Oops: 0000 [#5]
SMP
CPU: 0
EIP: 0060:[<ffffffff>] Not tainted
EFLAGS: 00010296 (2.6.6)
EIP is at 0xffffffff
eax: 0000000c ebx: ffffffff ecx: 00000000 edx: bfffda7c
esi: cf434f00 edi: ffffffff ebp: 00002000 esp: c27fff78
ds: 007b es: 007b ss: 0068
Process head (pid: 2331, threadinfo=c27fe000 task=c3226150)
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 fffffff7
bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000
00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70
Call Trace:
[<c0150612>] sys_read+0x42/0x70
[<c0103f8f>] syscall_call+0x7/0xb
Code: Bad EIP value.
```

Error message

0xffffffff points to nowhere

User-space address space if < 0xc000000

Bad address

Call stack

Baker/Wang, fsu.edu

Oops Messages

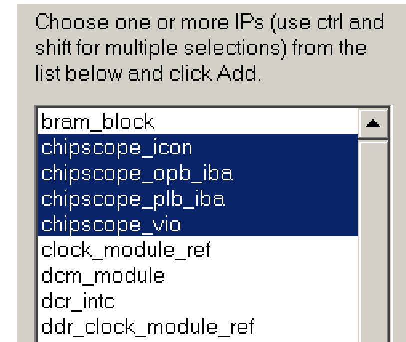
- Require `CONFIG_KALLSYMS` option turned on to see meaningful messages
- Other tricks
  - `0xa5a5a5a5` on stack → memory not initialized

Baker/Wang, fsu.edu

Xilinx Hardware & Software Debugging

Hardware Debugging Support

- ChipScope™ Pro tool cores are now included for adding to a Xilinx Platform Studio design
  - PLB IBA (Integrated Bus Analyzer)
  - OPB IBA
  - VIO (Virtual I/O)

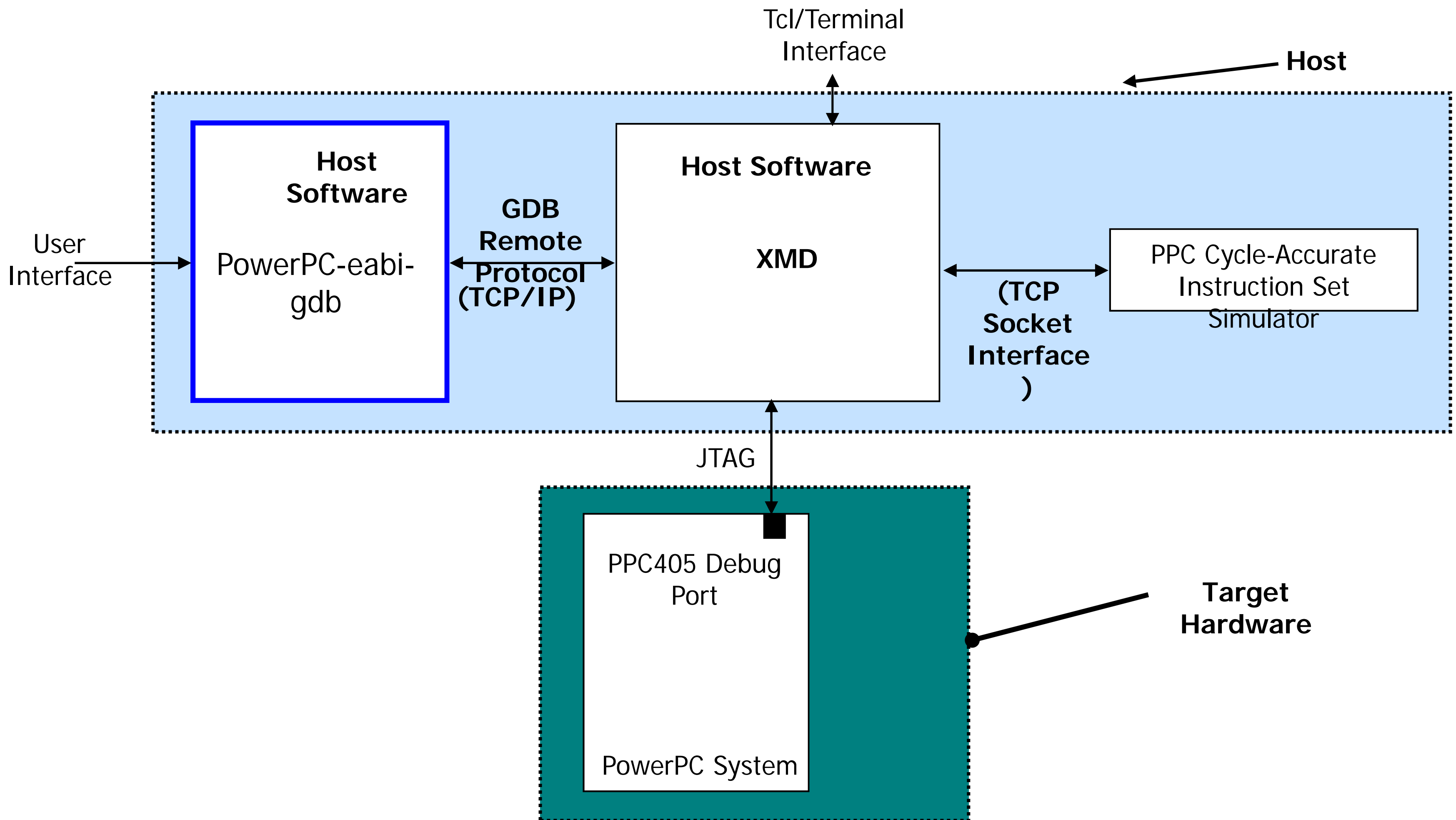


- ChipScope Pro tool available thru
- donation
- Enables co-debug of software with GNU gdb and hardware with ChipScope Analyzer

Software Debugging Support

- **EDK supports software debugging via:**
  - GNU Debugger (GDB) tools
    - Unified interface for debugging and verifying MicroBlaze™ and PowerPC™ systems
  - Xilinx Microprocessor Debugger (XMD)
    - Runs all of the hardware debugging tools and communicates with the hardware
  - GNU tools communicate with the hardware through XMD

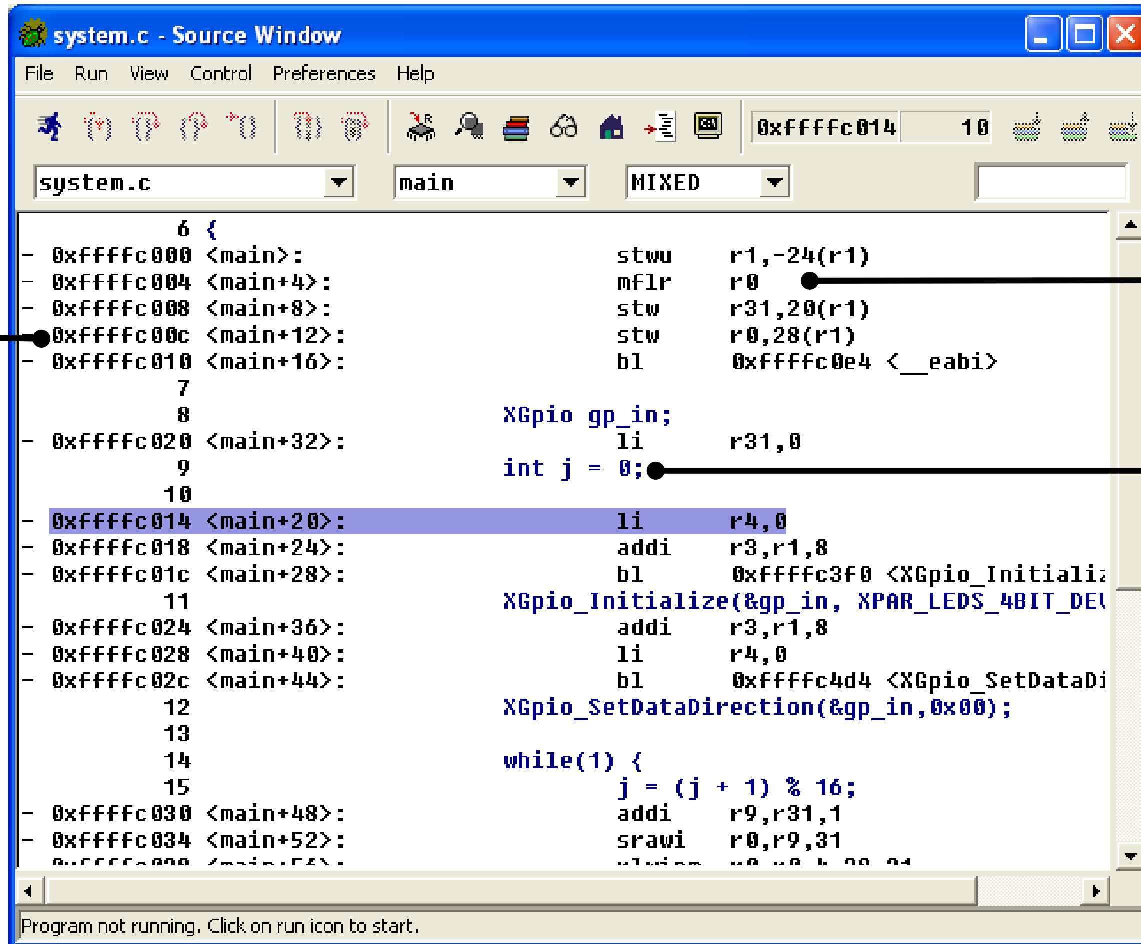
GDB Functionality



GDB Functionality

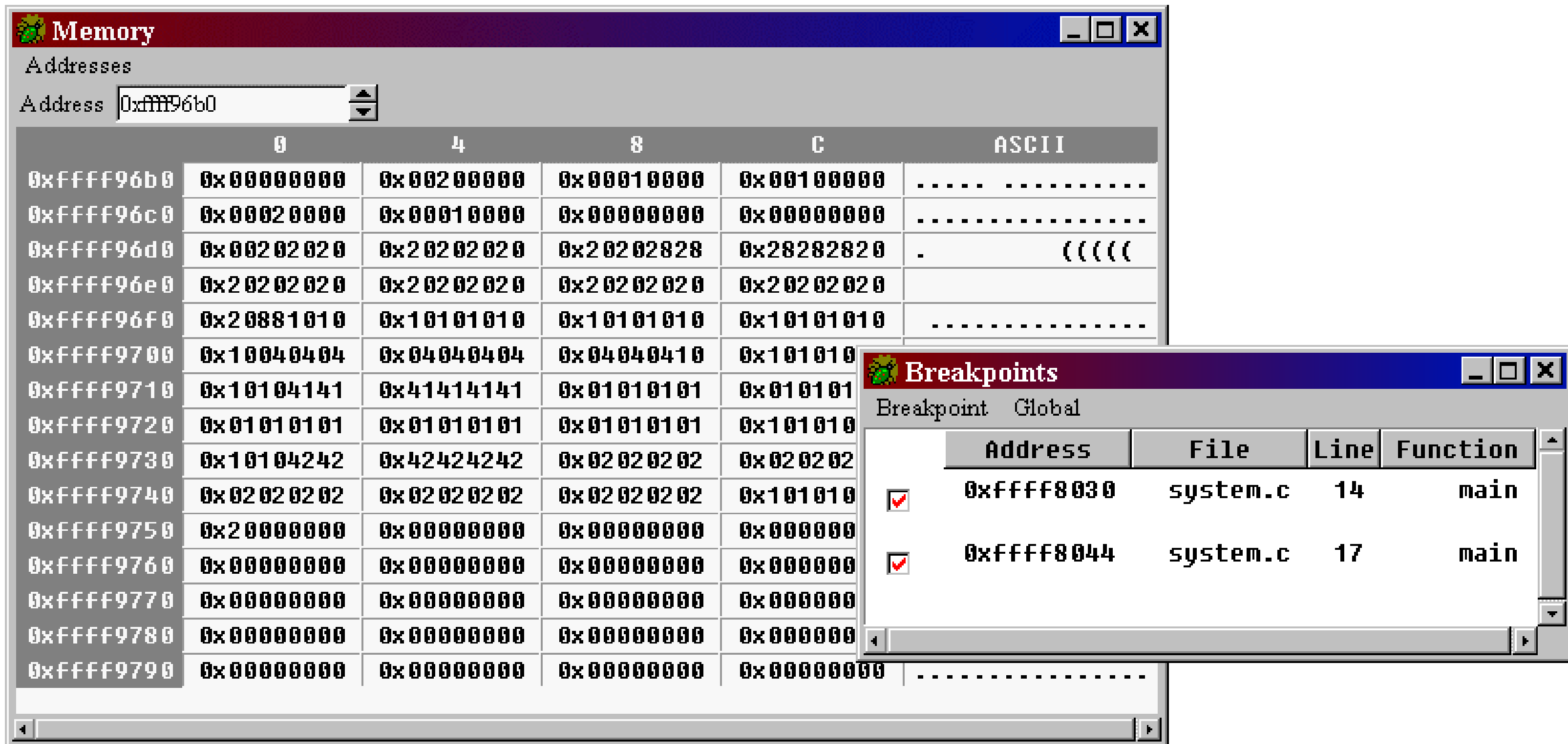
- GDB is a source-level debugger that helps you debug your program:
  - Start your program
  - Set breakpoints (make your program stop on specified conditions)
  - Examine what has happened, when your program encounters breakpoints
    - Registers
    - Memory
    - Stack
    - Variables
    - Expressions
  - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another
- You can use GDB to debug programs written in C and C++

GDB



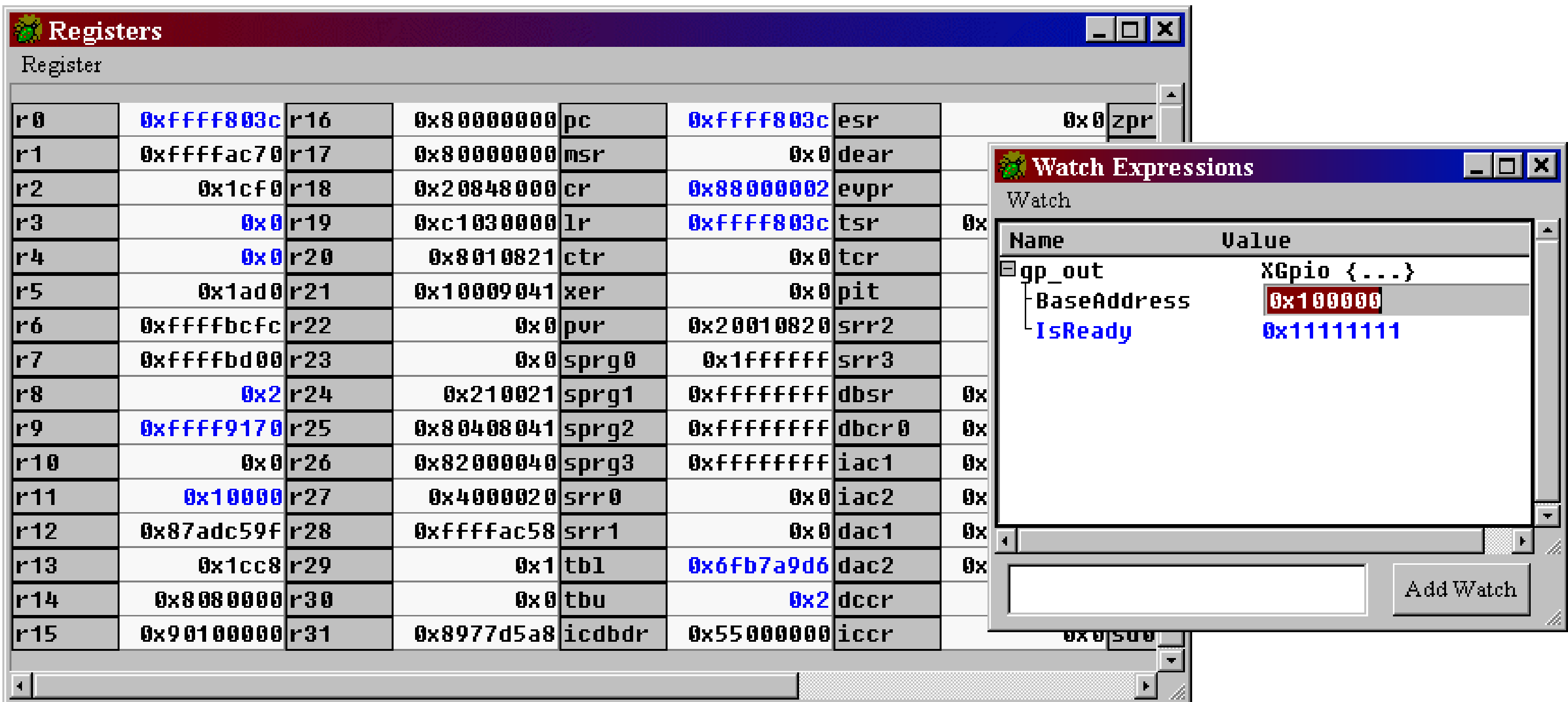
GDB Functionality

- Breakpoints can be enabled or disabled
- To change any memory value, simply double-click in a memory field

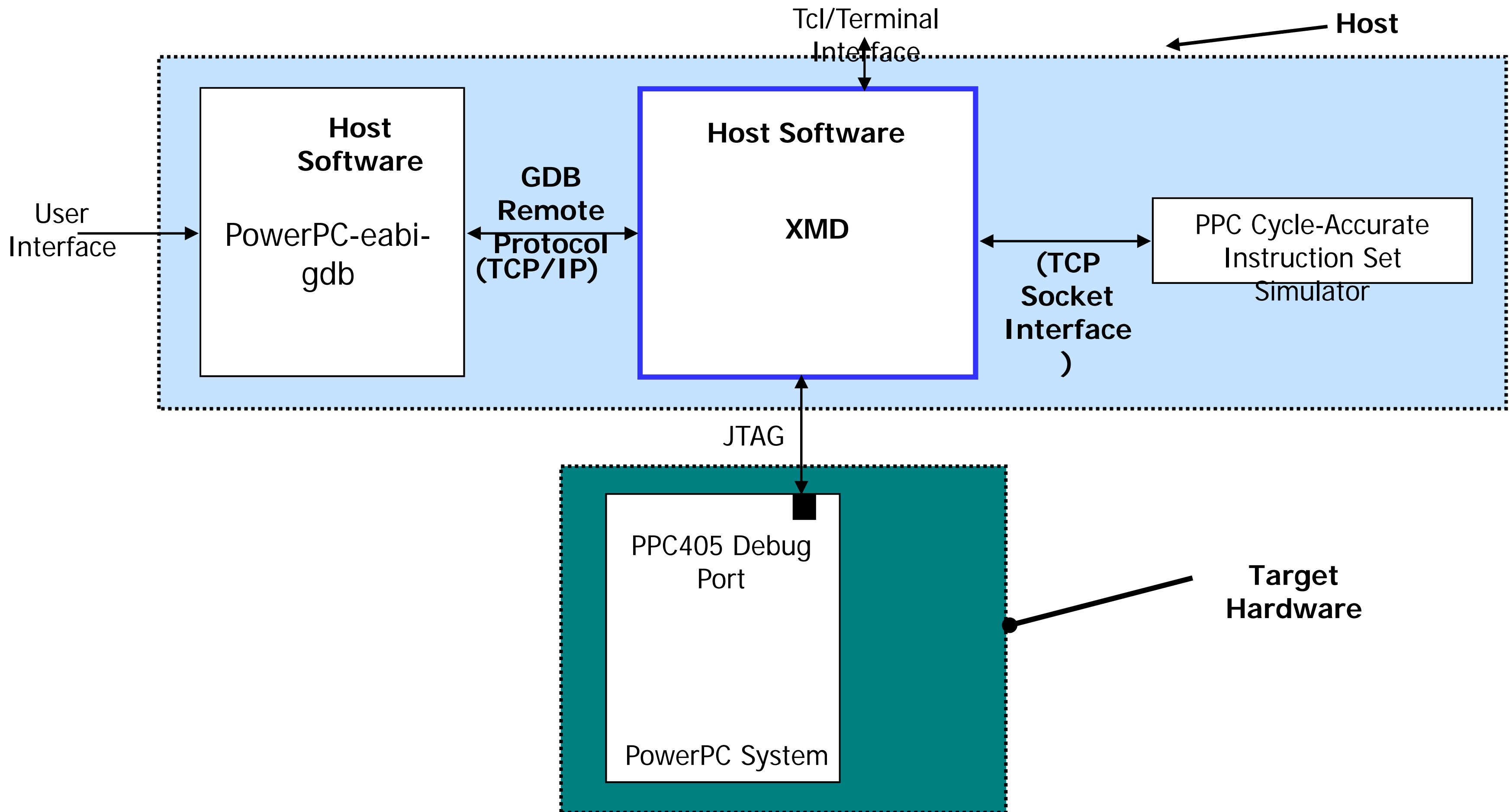


GDB Functionality

- **Blue represents registers that have changed**
- **To change any value, double-click in a field**



XMD Functionality

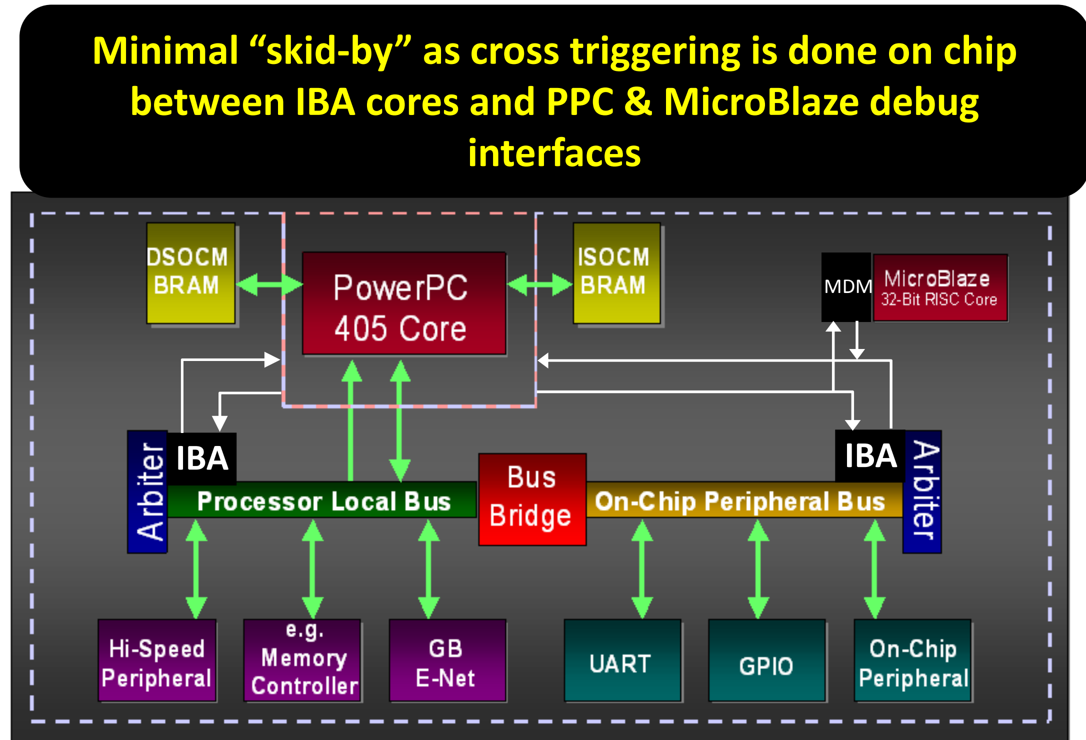


XMD Functionality

- Xilinx Microprocessor Debug (XMD) engine
  - A program that facilitates a unified GDB interface
  - A Tool command language(TCL) interface
- XMD supports debugging user programs on different targets
  - Cycle-accurate PowerPC™ processor instruction set simulator
  - PowerPC™ system on a hardware board
- GDB communicates with xmd by using the Remote TCP protocol and control the corresponding targets
- GDB can connect to xmd on the same computer or on a remote computer on the Internet

Simultaneous HW/SW Debug

- ChipScope™ Pro PLB & IBA cores in target
- ChipScope Pro Analyzer on host
- GDB debugger on host
- XMD supports simultaneous access over Xilinx parallel cables
- PLB/OPB IBA instantiation in XPS
  - Are treated like the peripheral cores



Set breakpoint in GDB – when hit → triggers ChipScope

Set trigger in ChipScope – when hit → halts CPU and debugger stops

Simultaneous HW/SW Debug

dummy.c - Source Window

File Run View Control Preferences Help

dummy.c main MIXED

```
      6 *(int*) 0x0000c204 = 0;
-    0xd8 <main+4>:          imm    0
-    0xdc <main+8>:          swi    r0, r0, -15868 // 0xc204
      7
      8 while (1) {
-    0xe0 <main+12>:         imm    0
-    0xe4 <main+16>:         addik  r5, r0, -15872 // 0xc200
      9    *(int*) 0x0000c200 = i;
-    0xe8 <main+20>:         lw     r4, r0, r1
-    0xec <main+24>:         sw     r4, r0, r5
     10   i++;
-    0xf0 <main+28>:         lw     r3, r0, r1
-    0xf4 <main+32>:         addik  r3, r3, 1
-    0xf8 <main+36>:         sw     r3, r0, r1
     11 }
-    0xfc <main+40>:         bri    -20     // 0xe8 <main+20>
```

Program is running.

Active trigger when addr bus = 0xC200

ChipScope Pro Analyzer [system]

Trigger out signal from IBA to CPU debug halt signal in

XMD

Xilinx Parallel Cable

JTAG

MDM

MicroBlaze 32-Bit RISC Core

Flexible Soft IP

IBA

OPB

Arbiter

On-Chip Peripheral Bus

Backup

Remote Debugging

- The following are the steps that needs to be done to enable remote debugging
- Before you begin
- Make sure you compile your program with the "–g" option.
- Load the program into the ARM using wget/usb and give appropriate permissions
- Make sure a copy of the program exists in your working directory in the Linux environment accessible through one of the soc<num>.ece.utexas.edu machines.
- In the Linux environment on the board
- Find out the IP of the arm board using "ifconfig". Note the IP number you will need it.
- Run the following command
- gdbserver soc<num>.ece.utexas.edu:<portnumber> <programName>
- Where
- num can be 1,2 or 3
- portnumber is any port that is not being used, 2345 is one such port
- programName is the program that you are trying to debug
- This would have started the gdbserver.

- In the windows environment
- Using putty login into the machine specified while starting the gdbserver
- Navigate to directory that contains the program
- Start the arm-linux-gdb
- In the prompt execute command "symbol-file <programName>"
- Then "target remote <board_IP_Address>:<portnumber>"
  - Board_IP_Address: is the address found earlier using ifconfig
  - portNumber: needs to be same as used earlier
- This would link to the gdb server and you are ready to debug. Please note that the program would have already started execution, hence don't use the run command in the dgb. Also using the run command will kill the gdbserver and would require restarting.