Software Development Tools for Embedded Systems

Hesen Zhang

What Are Tools?

a handy tool makes a handy man

What Are Software Development Tools?

Outline

- Debug tools
  GDB practice
  Debug Agent Design
  Debugging with JTAG

- Analysis Tools
  Strace
  Mtrace
  Valgrind

- Dynamic Adaption Tool

  Background-Cyber
  Physical System

  Introduction to Kevoree

GDB debugging

The GDB package is open-source software that can be downloaded and configured for a specific architecture.

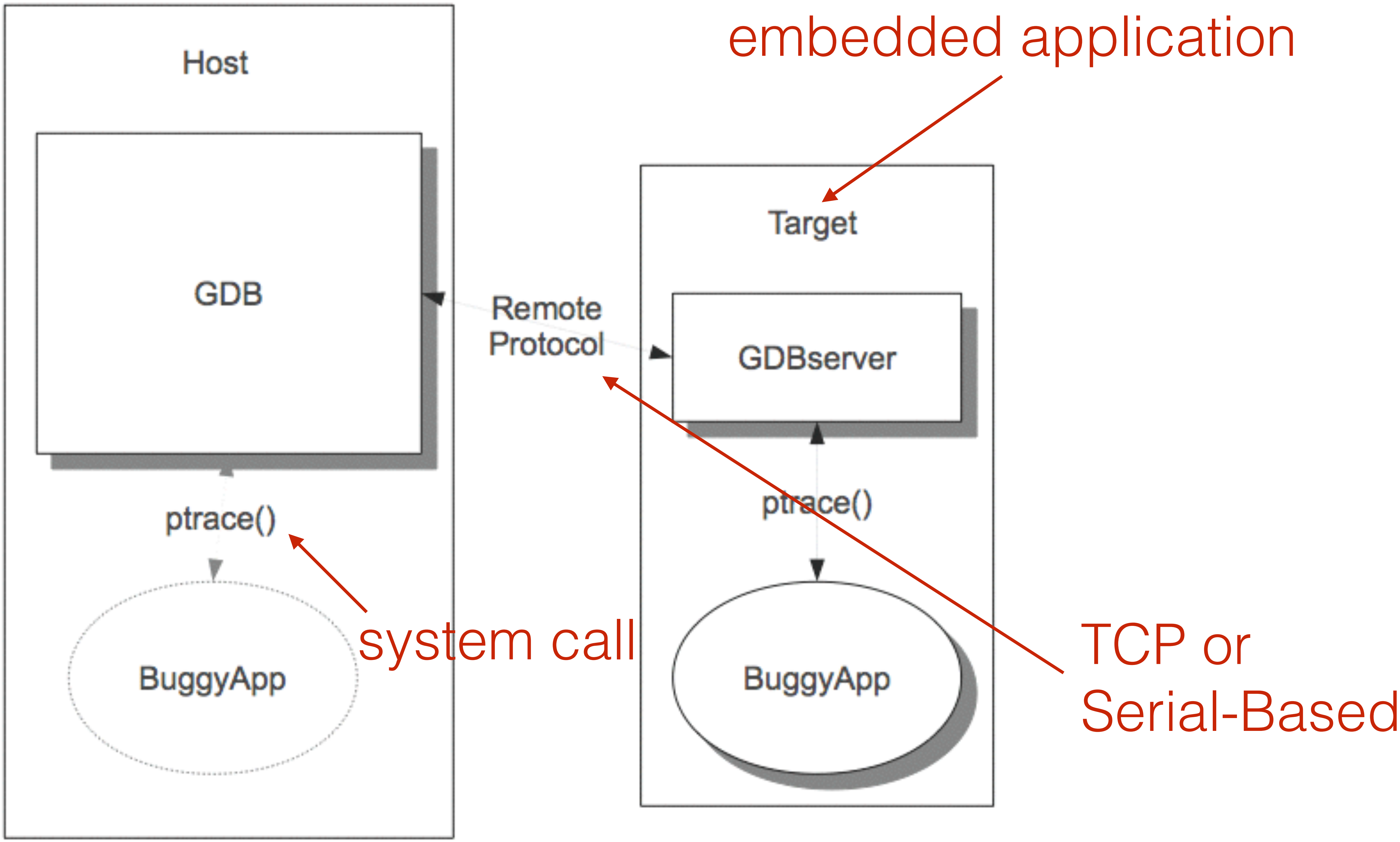
- single application (gdb) this is most commonly used in the case of debugging host applications

GDB debugging

The GDB package is open-source software that can be downloaded and configured for a specific architecture.

- single application (gdb) this is most commonly used in the case of debugging host applications
- When debugging embedded applications, the gdb-gdbserver mode is what normally should be used

GDBserver



Host
GDB
ptrace()
BuggyApp
Remote
Protocol
Target
GDBserver
ptrace()
BuggyApp
embedded application
system call
TCP or
Serial-Based

Examining Data

demo.c

```c
int main()
{
   int a[] = {1,2,3};
   return 0;
}
```

```
$ gcc -g demo.c -o demo
$ gdb arrays
(gdb) break main
(gdb) run
(gdb) next
```

```
(gdb) print a
$1 = {1, 2, 3}

(gdb) ptype a
type = int [3]

(gdb) ptype &a
type = int *

(gdb) whatis main
type = int (void)
```

```
(gdb) x/12xb &a
0x7fff5fbff56c: 0x01  0x00  0x00  0x00  0x02  0x00  0x00  0x00
0x7fff5fbff574: 0x03  0x00  0x00  0x00
```

Stack Frame Trace

demo.c

```c
#include <stdio.h>

int func(int n)
{
   int sum=0,i;
   for(i=0; i<n; i++){
      sum+=i;
   }
   return sum;
}

main(){
   int i;
   long result = 0;
   for(i=1; i<=100; i++){
      result += i;
    }

 printf("result[1-100] = %d \n", result );
 printf("result[1-250] = %d \n", func(250));
}
```

```
Breakpoint 1, main () at gdbtest.c:17
17    long result = 0;
(gdb) n
…
(gdb) c
Continuing.
result[1-100] = 5050

Breakpoint 2, func (n=250) at gdbtest.c:5
5     int sum=0,i;
(gdb) bt
#0  func (n=250) at gdbtest.c:5
```

Changing the program

Change the value of a local or global variable: assign 11 to variable "i":

```
(gdb) set variable i = 11
```

Change the memory: set value 37 to the memory 0xbfc45400, converted to int:

```
(gdb) set {int}0xbfc45400 5 37
```

Change the value of a register:

```
(gdb) set $r050310
```

Modify the execution address: the program counter is modified. The next run control command (r, c, s, n) will execute from this new program counter address:

```
(gdb) set $pc=0x80483a7
(gdb) set $pc=&compute_data
```

Continue at a different address: resume execution at the specific line or at specific address

```
(gdb) jump data.c:19
Continuing at 0x80483a7
```

Execute a function:

```
(gdb) call func(3)
```

Return a function:

```
(gdb) return 1
```

Segmentation Fault and Core Dump

What is a core dump?

— binary file record info when operating system terminated abnormally

e.g.: core.PID

Enable core dumps

```
$ ulimit -c unlimited
```

```
#include <stdio.h>
int main(){
   func();
   return 0;
}

int func(){
   char* p = 1;
   *p = ‘a’;
}
```

```
$ll #to see the core.pid file#
$gdb ./a.out ./core.7369
Core was generated by `./a.out'.
Program terminated with signal 11, Segmentation fault.
#0  0x080483b8 in fund () at ./test.c:10
10        *p = 'a';
…
…
(gdb) where
#0  0x080483b8 in func () at ./test.c:10
#1  0x0804839f in main () at ./test.c:4
(gdb)
```

Debug agent design

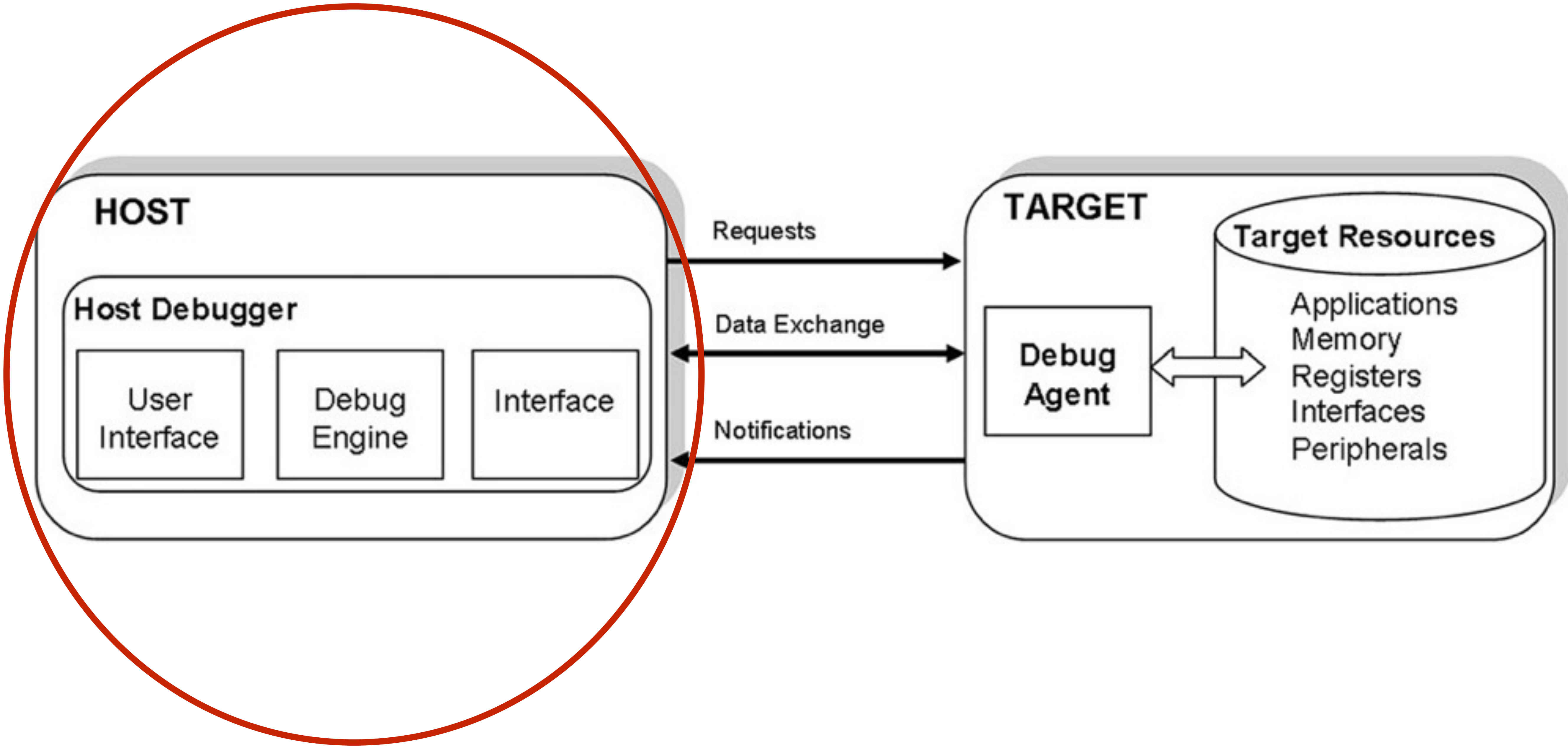
Scenarios where GDB cannot run on the target?

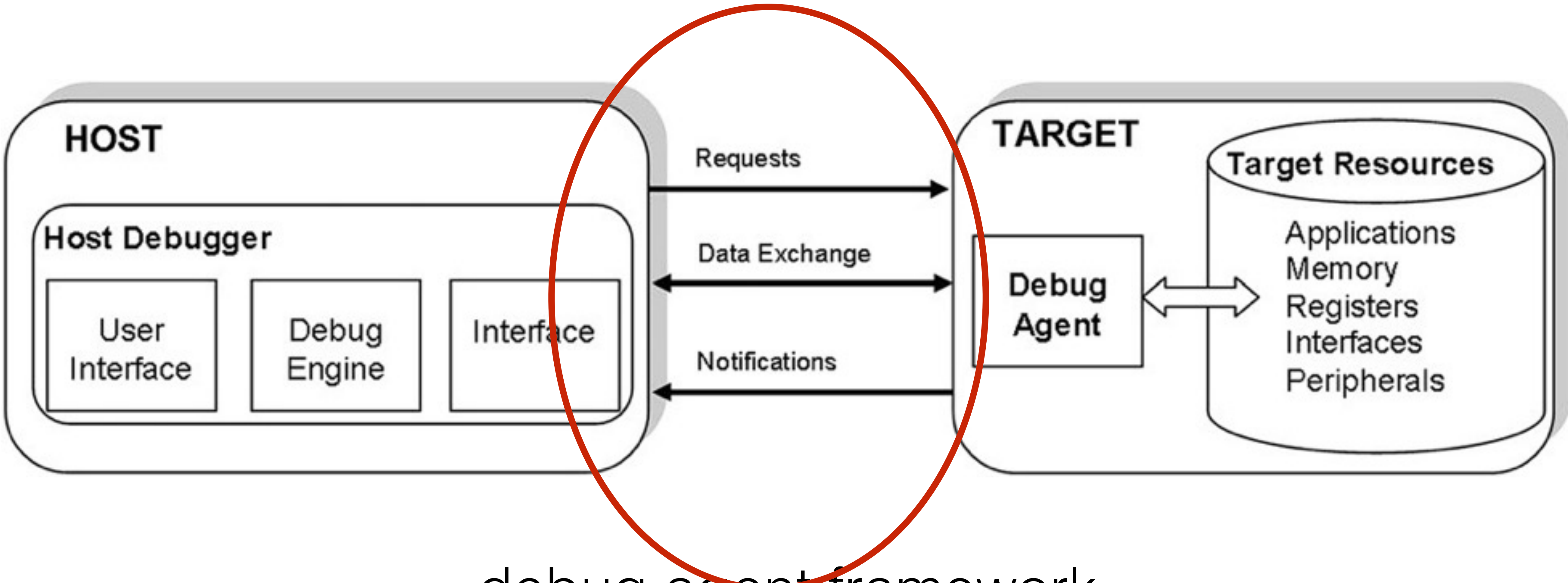
- Special Requirements
- Limited Resources

Examples:

- No Linux Operating System
- No Serial or Ethernet interface
- Not allow porting the debug agent

Debug agent design





debug agent framework

Simple Communication Protocol

Control commands

- pause the application
- continue the application

GDB remote protocol
TCP, Serial

Data exchange commands

- retrieve the application context:
  (registers, stack, application specific data);
- read and write memory.

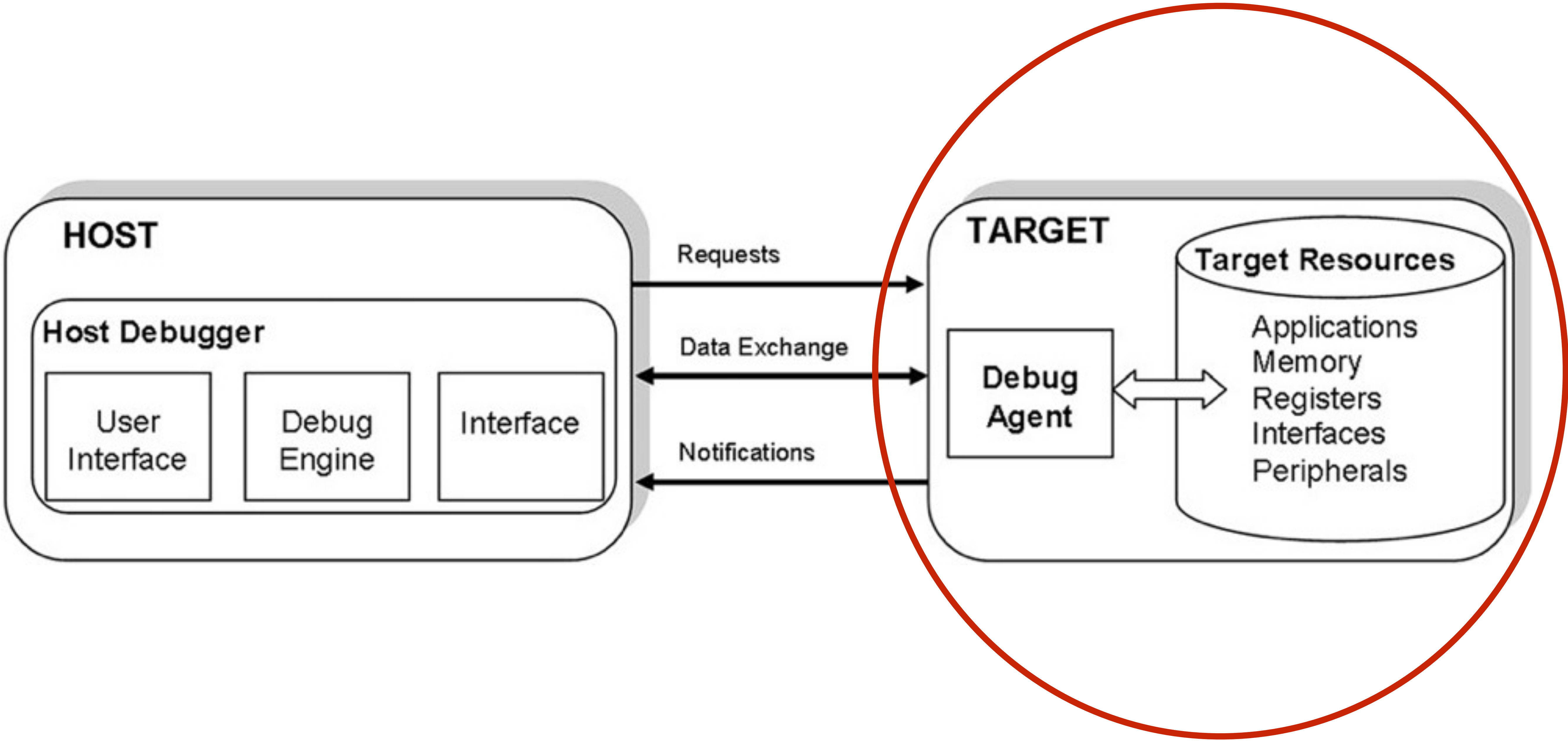
For Target Debug Agent

Physical Interface for communication with host:

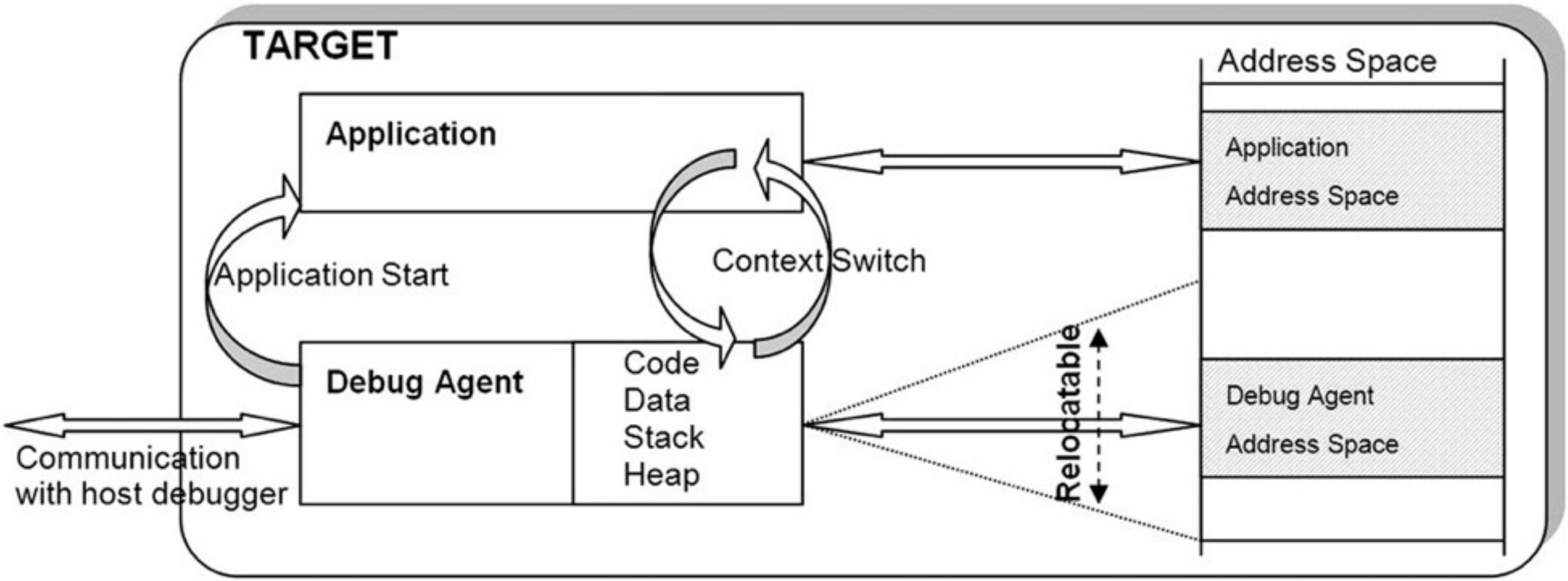
- Serial
- Ethernet
- USB
- Others

The debug agent implements the interrupt handler on the receive side of the interface and this interrupt is used as a debug event trigger.

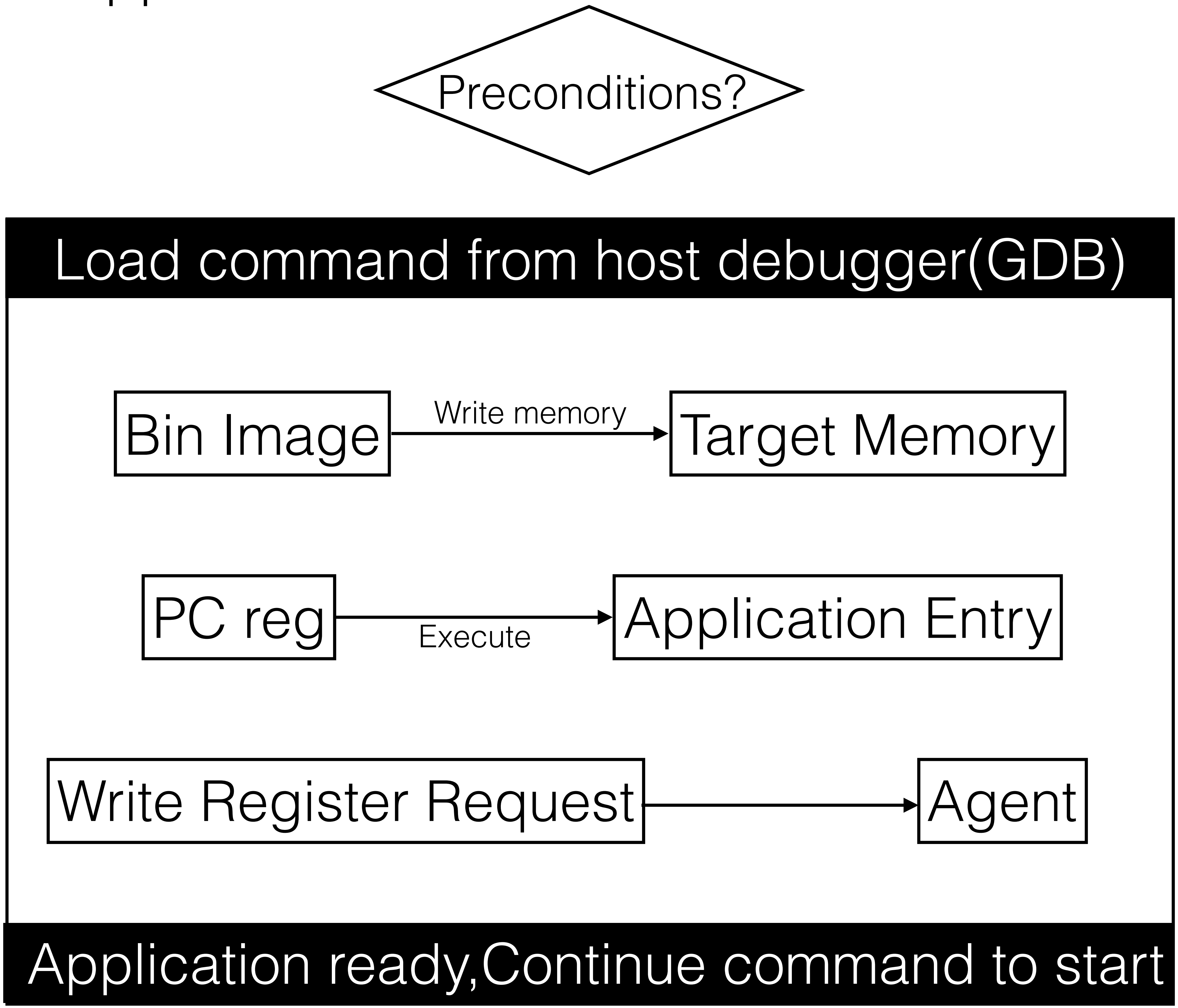
Debug agent design



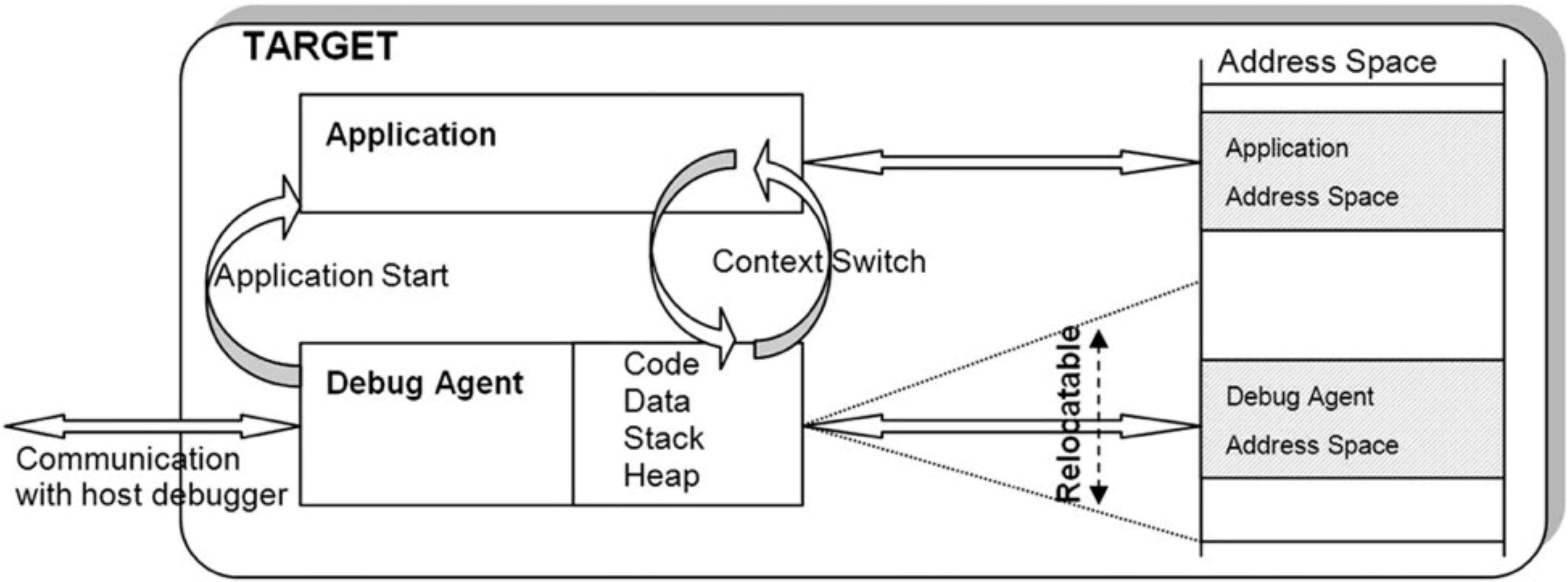
Debug agent design



Start the application



Debug agent design



Context Switch

Context?

Typically, general-purpose registers, Program Counter, Stack Register, and Link Register

Context Switch?

Saving and restoring of the application context

Context Switch

1. Triggered by a debug event
2. Save the application context
3. Save the address of instruction to resume
4. Initialize the stack for the debug agent
5. Interrupt handling while debug agent is running
6. The execution is passed to high level handler of debug agent

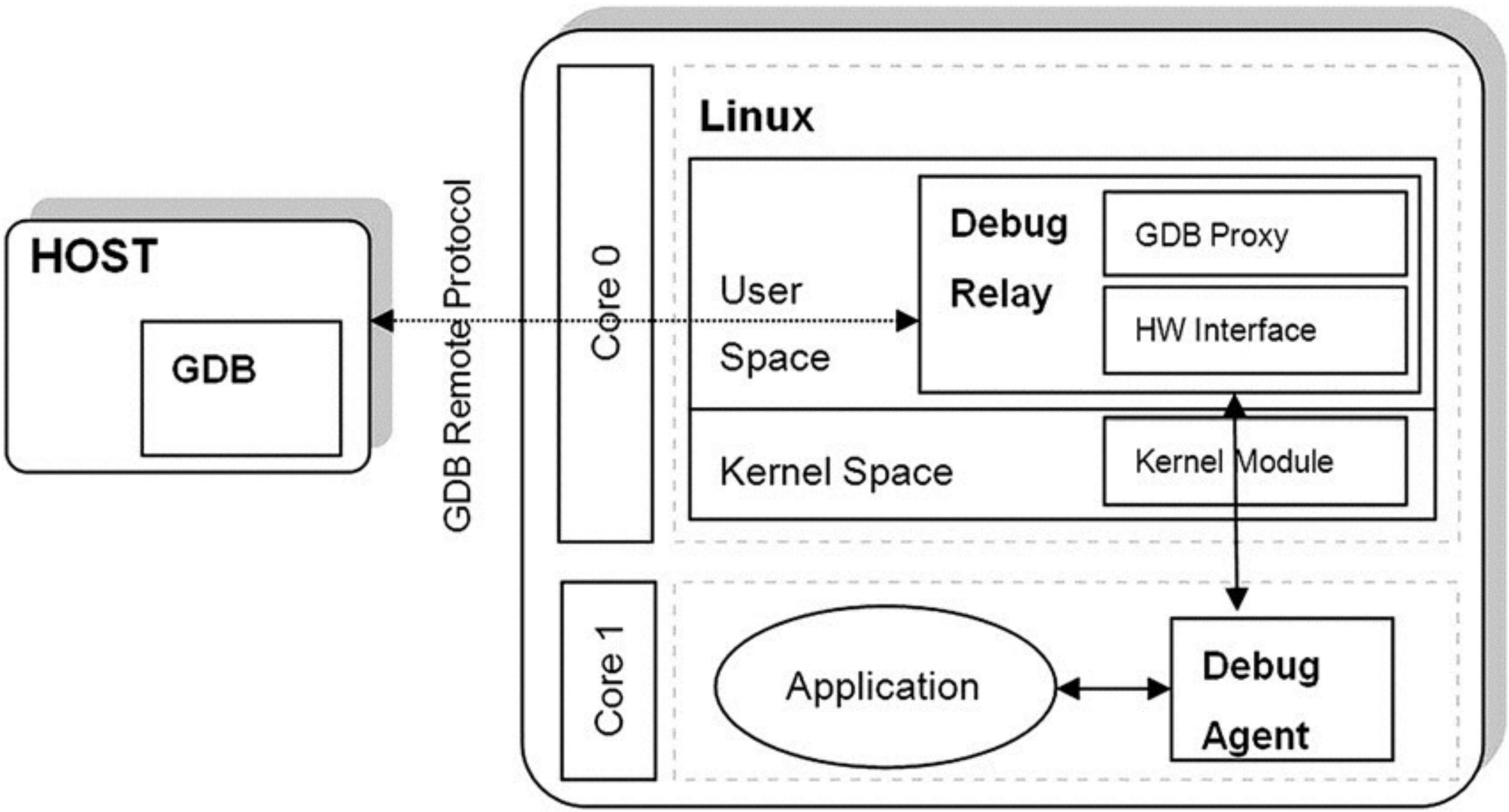
Context Switch

Do basic debug functions:

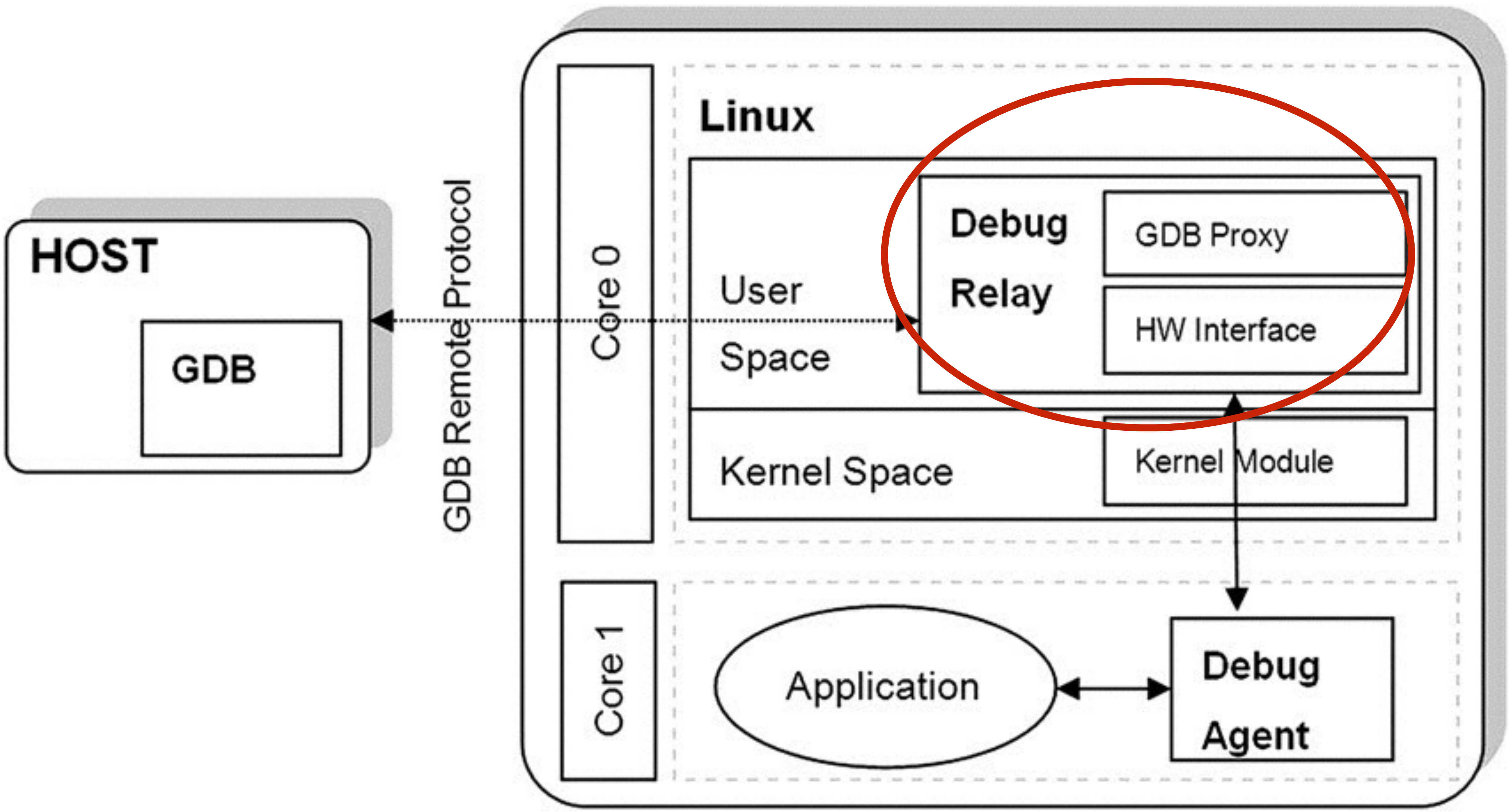
- Read and Write Registers
- Read and Write Memory and Stack
- Breakpoints
- Run Control

Restore context

Multicores

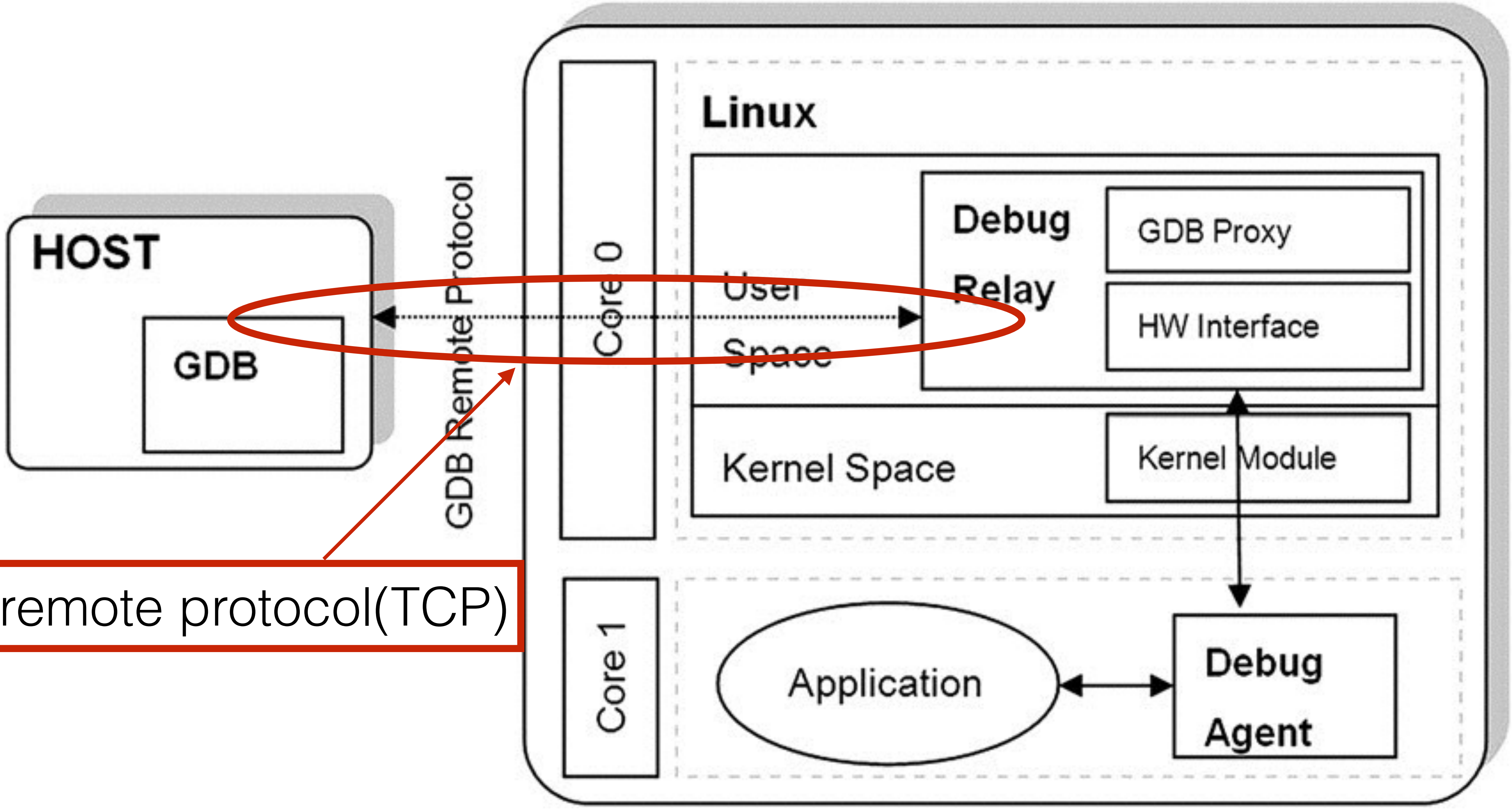

HOST
GDB
GDB Remote Protocol
Core 0
Linux
User
Space
Debug
Relay
GDB Proxy
HW Interface
Kernel Space
Kernel Module
Core 1
Application
Debug
Agent

Multicores



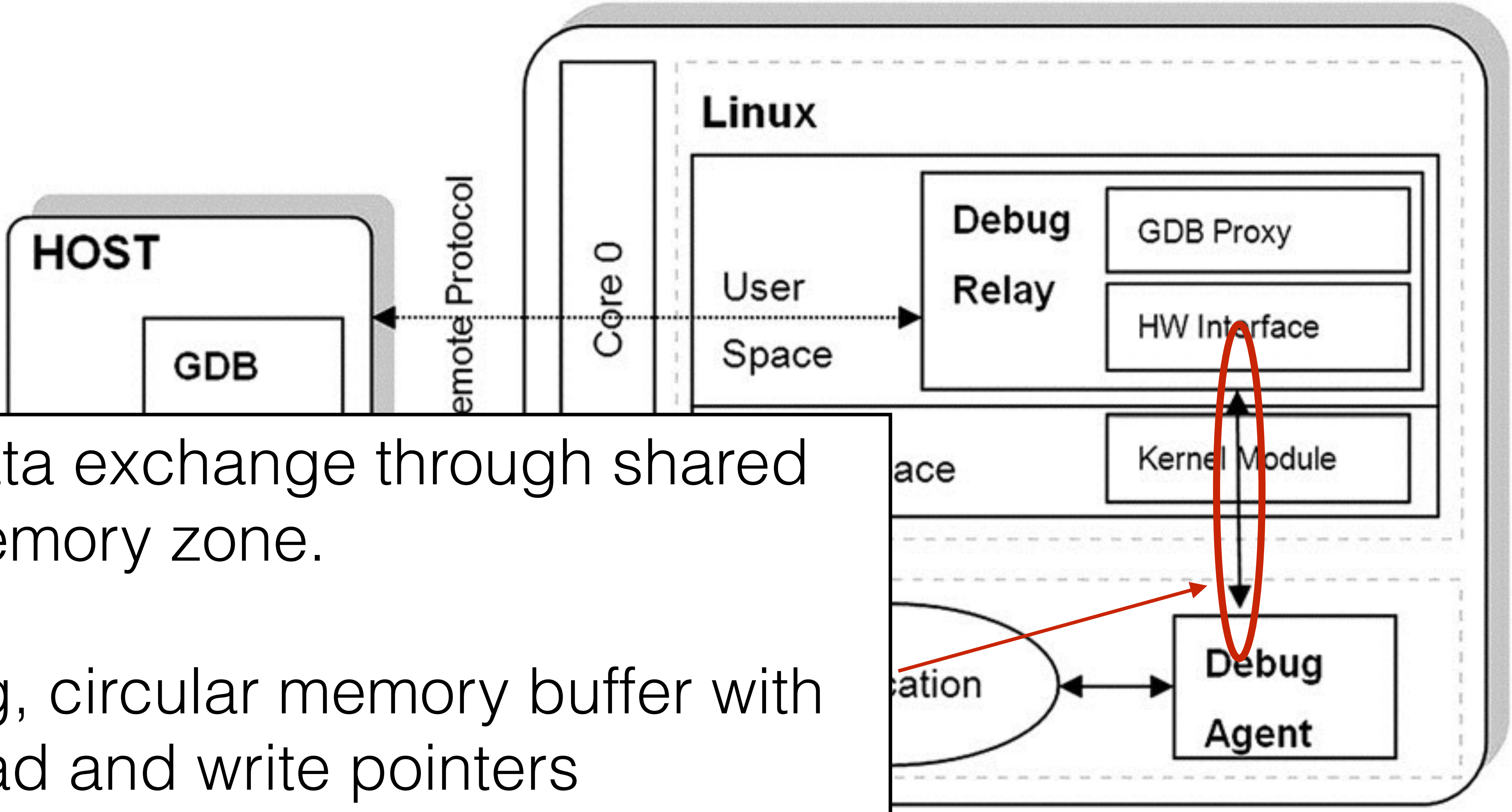
HOST
GDB
GDB Remote Protocol
Core 0
Linux
User
Space
Debug
Relay
GDB Proxy
HW Interface
Kernel Space
Kernel Module
Core 1
Application
Debug
Agent

Multicores



GDB remote protocol(TCP)

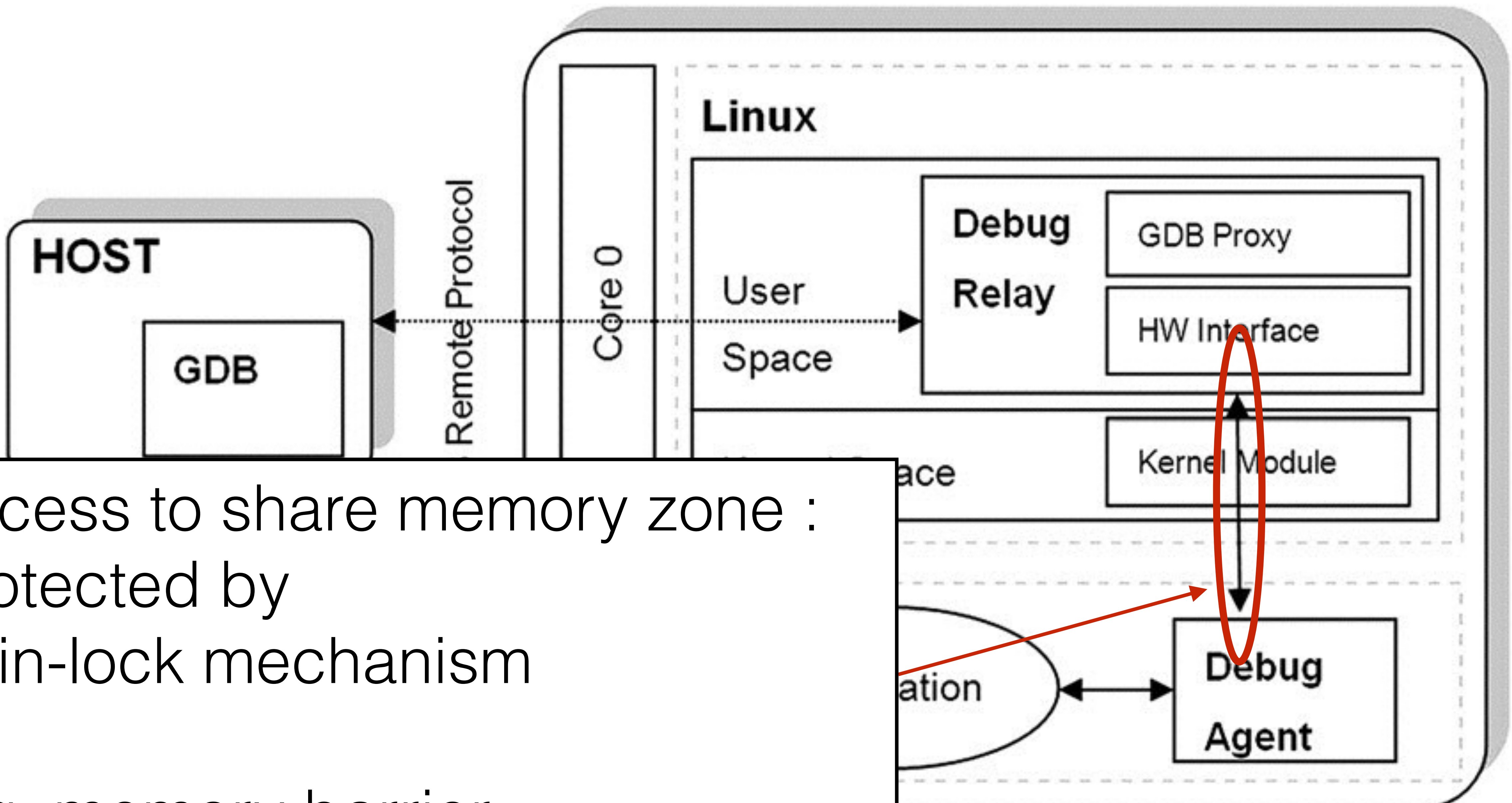
Multicores



Data exchange through shared memory zone.

e.g, circular memory buffer with read and write pointers

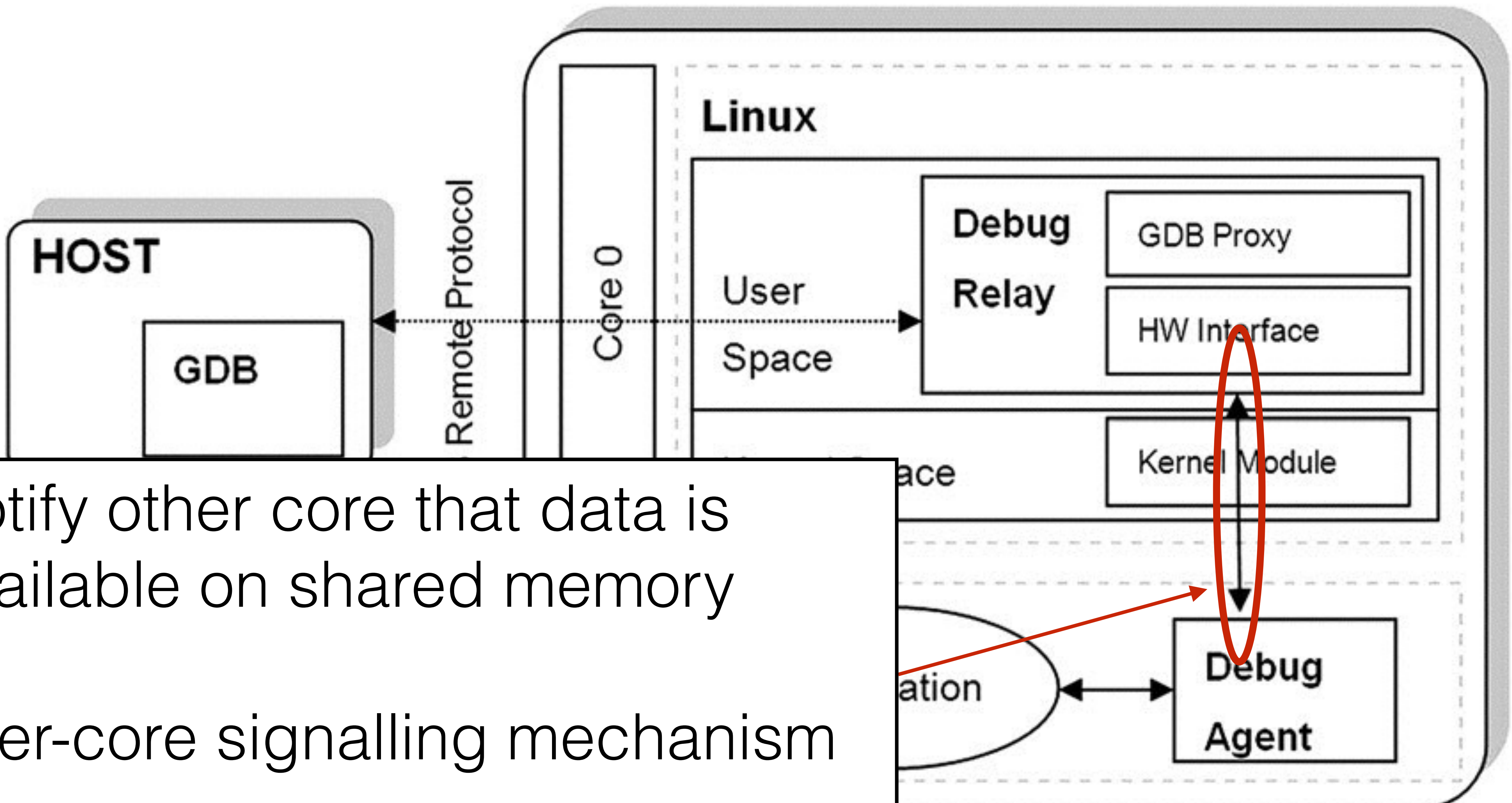
Multicores



access to share memory zone :
protected by
spin-lock mechanism

e.g. memory barrier

Multicores

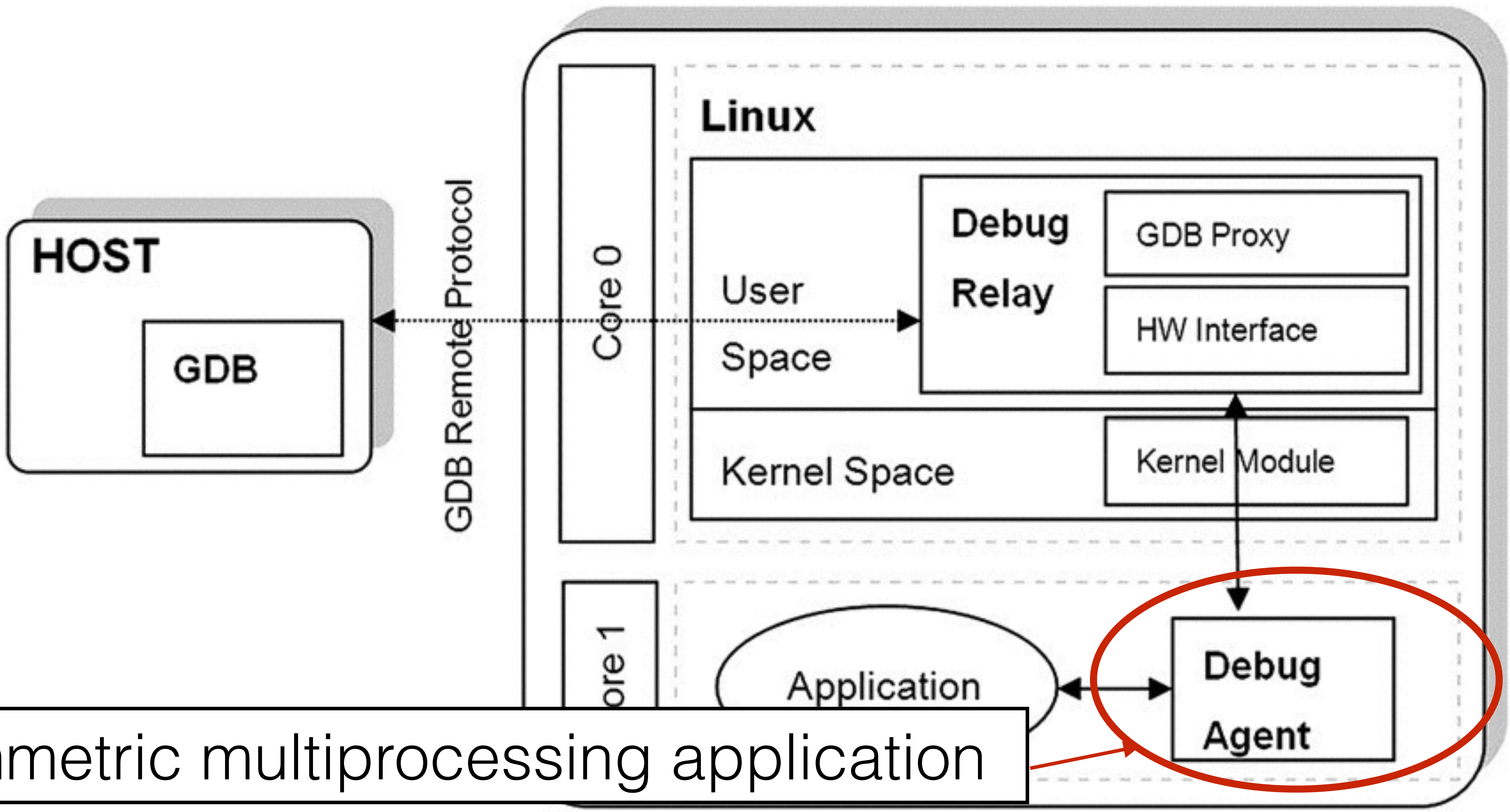


Notify other core that data is available on shared memory

Inter-core signalling mechanism

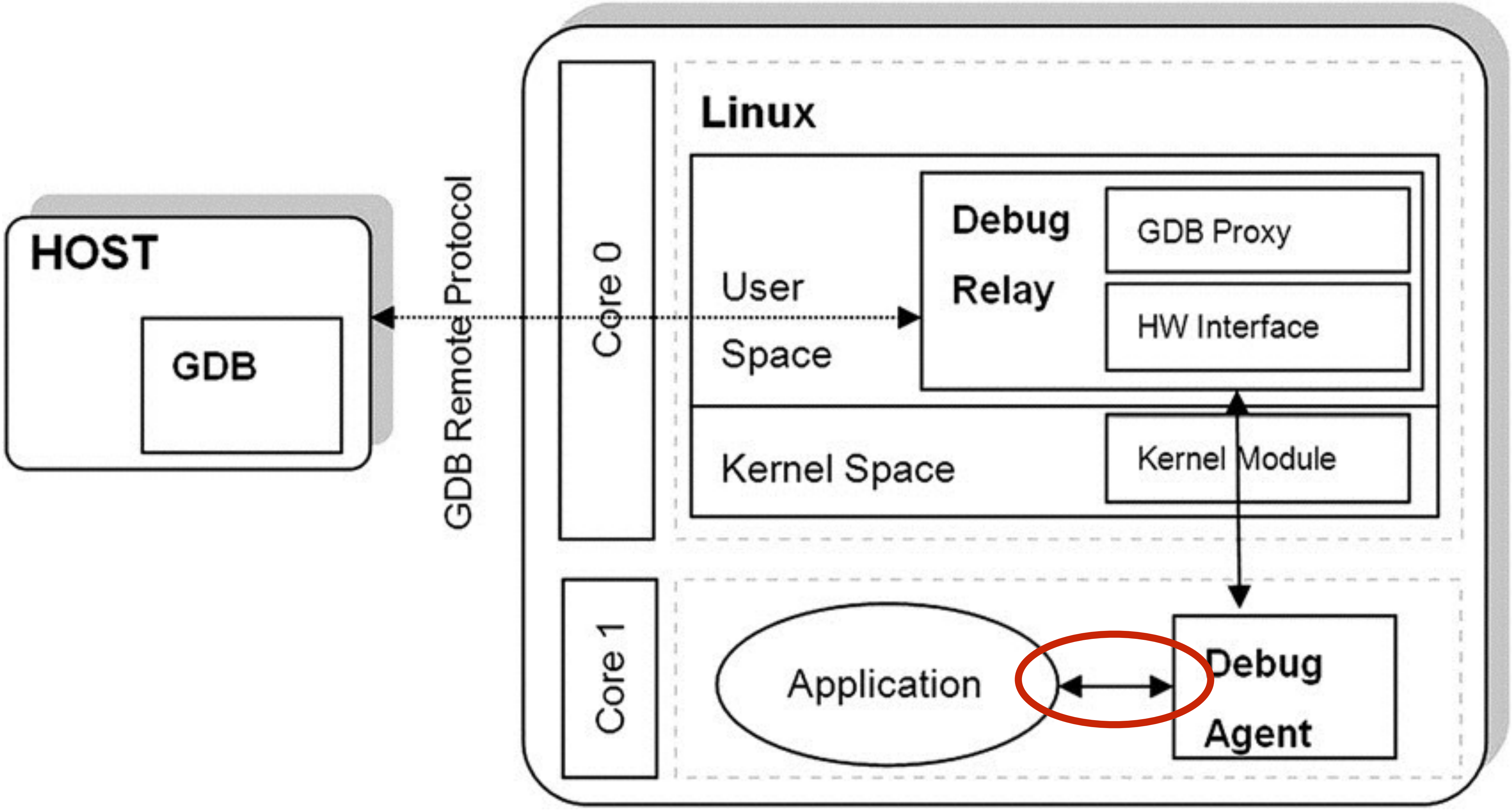
e.g: an inter-core interrupt based on the debug interrupt

Multicores



Symmetric multiprocessing application

Multicores


HOST
GDB
GDB Remote Protocol
Core 0
Linux
User
Space
Debug
Relay
GDB Proxy
HW Interface
Kernel Space
Kernel Module
Core 1
Application
Debug
Agent

Start the debug agent

Boot loader/Monitor program
e.g. u-boot

Debug agent binary copy to non-volatile memory
e.g. booting from flash memory

Multicore scenario: boot from Linux

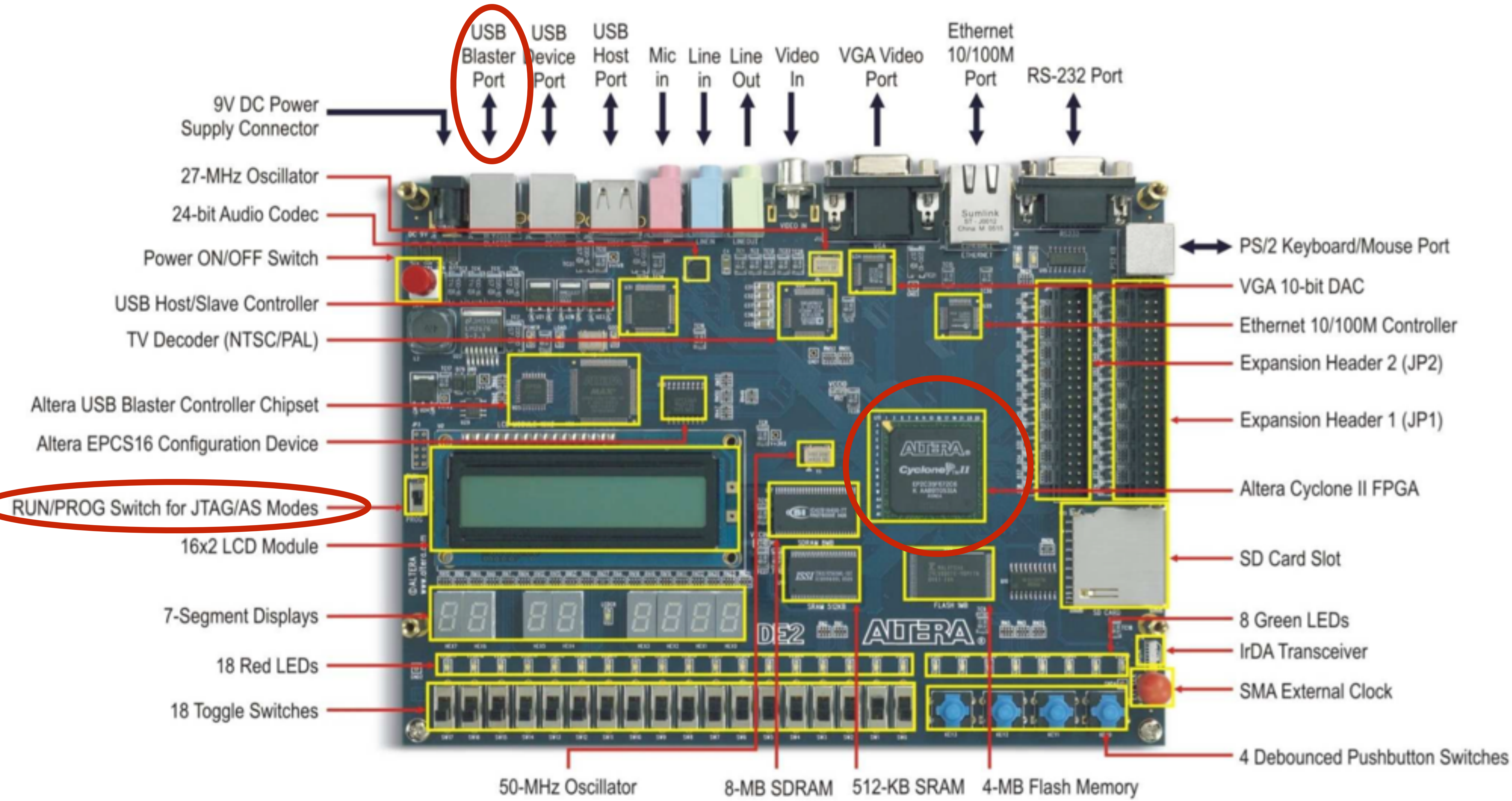
- copy agent binary into memory
- linux application to access the memory location
- linux tool convert ELF to bin
- start is similar to how Linux start for the secondary core

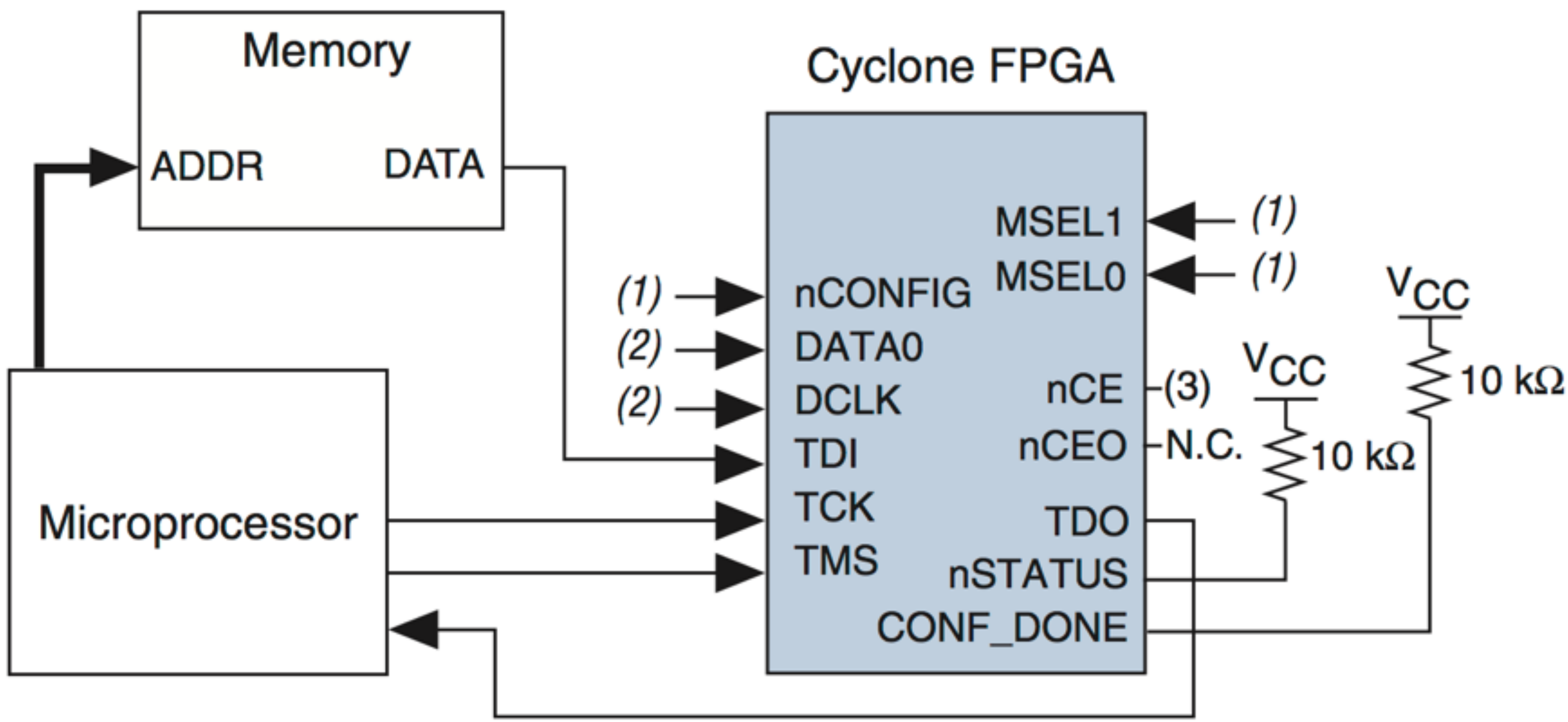
JTAG probe

Debugging using JTAG

- JTAG — Joint Test Action Group
- Circuits and boundary scan testing,
  debugging embedded systems including
  processors and FPGA circuits,
  data transfer into internal flash memory of circuits,
  flash programming, trace and analysis.
- Physically an external device

Synthesis Nios II into FPGA on DE2 board





JTAG Configuration of Cyclone FPGAs with a Microprocessor

Implementation Structure

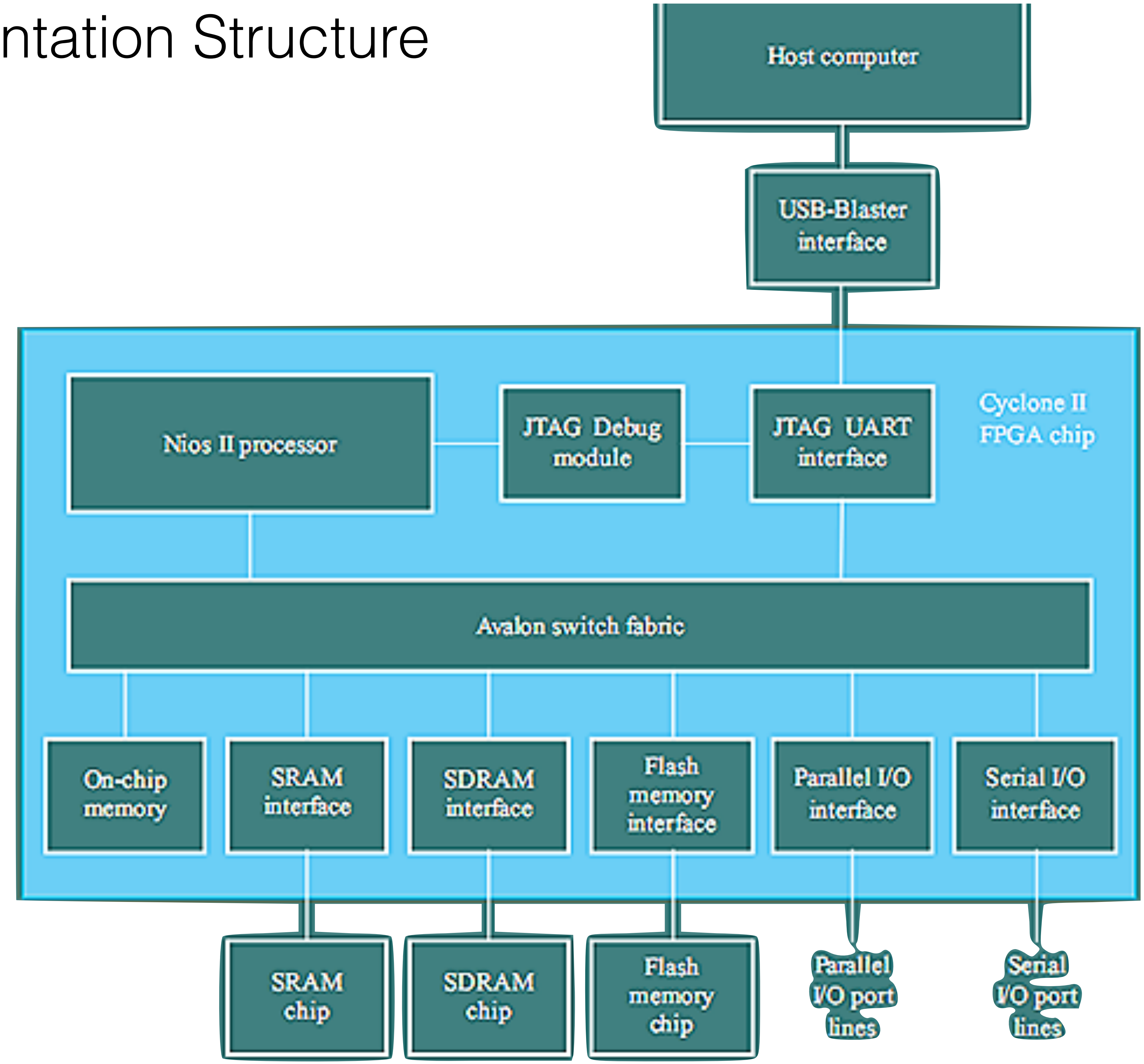


Figure 1. A Nios II system implemented on the DE2 board.

Benefits of using JTAG

Short development time!

Compare with GDB:

- JTAG for initial board bring-up, early application debug and when the debug agent software is not available, typically for bare board applications with no operating system;

- debug agent for high-level debug, typically after some operating system services are available for the debug agent. A common case is Linux user-space application debug using GDB/GDBserver.

Analysis Tools

- Strace
  examine system calls
- Mtrace
  investigate dynamic memory allocation
- Valgrind
  memory debugging,
  memory leak detection,
  and profiling

Strace

- Linux shell command that traces the execution of a program by intercepting and recording the system calls which are called by a process and the signals which are received by a process.

- What is A System Call?
  A system call is a special function used by a program to request a service from the kernel.

  read(), write(), open(), execve(), connect(), futex()…

Strace

Calling the strace followed by the name of the program:

```
$  strace ls
```

Example Result:

```
execve("/bin/ls", ["ls"], [/* 21 vars */]) = 0
brk(0)                                   = 0x8c31000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb78c7000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=65354, ...}) = 0
...
...
...
```

Other Options: -e -o -p -t -r -c

Mtrace

- “Memory Trace”
- investigate dynamic memory allocation
  - memory allocated that has not been deallocated (so called memory leaks)
  - deallocating not allocated memory
- included in the GNU C library
- consists of two main parts:
  - the runtime routine
  - the static routine
- Limitation: C++ Application

Mtrace

Example Code: test_mtrace.c

```
#include <stdio.h>
#include <stdlib.h>
#include <mcheck.h>

int main(){
   int *a, *b;
   char *c;

   a = (int *)malloc(sizeof(int));

   mtrace();
   b = (int*)malloc(sizeof(int));

   c = (char*)malloc(100*sizeof(char));
   free(a);
   muntrace();
   free(c);
   return 1;
}
```

Mtrace

Compile code, -g Option must be included:

```
$  gcc -g  test_mtrace.c  -o test_mtrace
```

Set the log export path, run the elf:

```
$  export MALLOC_TRACE= test_mtrace.log
./test_mtrace
```

Analyze the log:

```
$  mtrace test_mtrace  test_mtrace.log
```

Example Result:

```
- 0x0000000001f3e010 Free 4 was never alloc'd /home/demo/test_mtrace.c:16

Memory not freed:
-----------------
Address              Size    Caller
0x0000000001f3e480   0x4     at /home/demo/test_mtrace.c:12
0x0000000001f3e4a0   0x64    at /home/demo/test_mtrace.c:14
```

Valgrind

- open-source free software, GNU
- x86, amd64, ppc32, ppc64 and s390x
- common used tools:
  - Memcheck
  - Cachegrind
  - Callgrind
  - Helgrind
  - Massif
  - Extension

Valgrind

Compile code, -g Option must be included:

```
$  gcc -g  test_valgrind.c  -o test_valgrind
```

Run valgrind with corresponding tool:

```
$ valgrind --tool=tool_name program_name
```

Example Code: test_valgrind.c

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
   int *a, b, *c;

   a = (int*)malloc(sizeof(int));
   *a = b;
   printf("*a = %d\n", *a);
   c = (int*)malloc(10*sizeof(int));
   printf("c[11] = %d\n",c[11]);
   return 1;
}
```

$ valgrind —tool=memcheck –leak-check=yes ./test_valgrind

memcheck

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
   int *a, b, *c;

   a = (int*)malloc(sizeof(int));
   *a = b;
   printf("*a = %d\n", *a);
   c = (int*)malloc(10*sizeof(int));
   printf("c[11] = %d\n",c[11]);
   return 1;
}
```

```
==20594== Use of uninitialised value of size 8
==20594==    at 0x4E7A49B: _itoa_word (_itoa.c:195)
==20594==    by 0x4E7C4E7: vfprintf (vfprintf.c:1596)
==20594==    by 0x4E85298: printf (printf.c:35)
==20594==    by 0x40057C: main (test_valgrind.c:9)
==20594==
==20594== Conditional jump or move depends on uninitialised value(s)
==20594==    at 0x4E7A4A5: _itoa_word (_itoa.c:195)
==20594==    by 0x4E7C4E7: vfprintf (vfprintf.c:1596)
==20594==    by 0x4E85298: printf (printf.c:35)
==20594==    by 0x40057C: main (test_valgrind.c:9)
```

```
$ valgrind —tool=memcheck –leak-check=yes ./test_valgrind
```

memcheck

```c
#include <stdio.h>
#include <stdlib.h>

int main(){
   int *a, b, *c;

   a = (int*)malloc(sizeof(int));
   *a = b;
   printf("*a = %d\n", *a);
   c = (int*)malloc(10*sizeof(int));
   printf("c[11] = %d\n",c[11]);
   return 1;
}
```

```
==20594== Invalid read of size 4
==20594==    at 0x400593: main (test_valgrind.c:11)
==20594==  Address 0x51f00bc is 4 bytes after a block of size 40 alloc'd
==20594==    at 0x4C2B6CD: malloc (in /usr/lib/valgrind/
vgpreload_memcheck-amd64-linux.so)
==20594==    by 0x400586: main (test_valgrind.c:10)
==20594==
```

$ valgrind —tool=memcheck –leak-check=yes ./test_valgrind

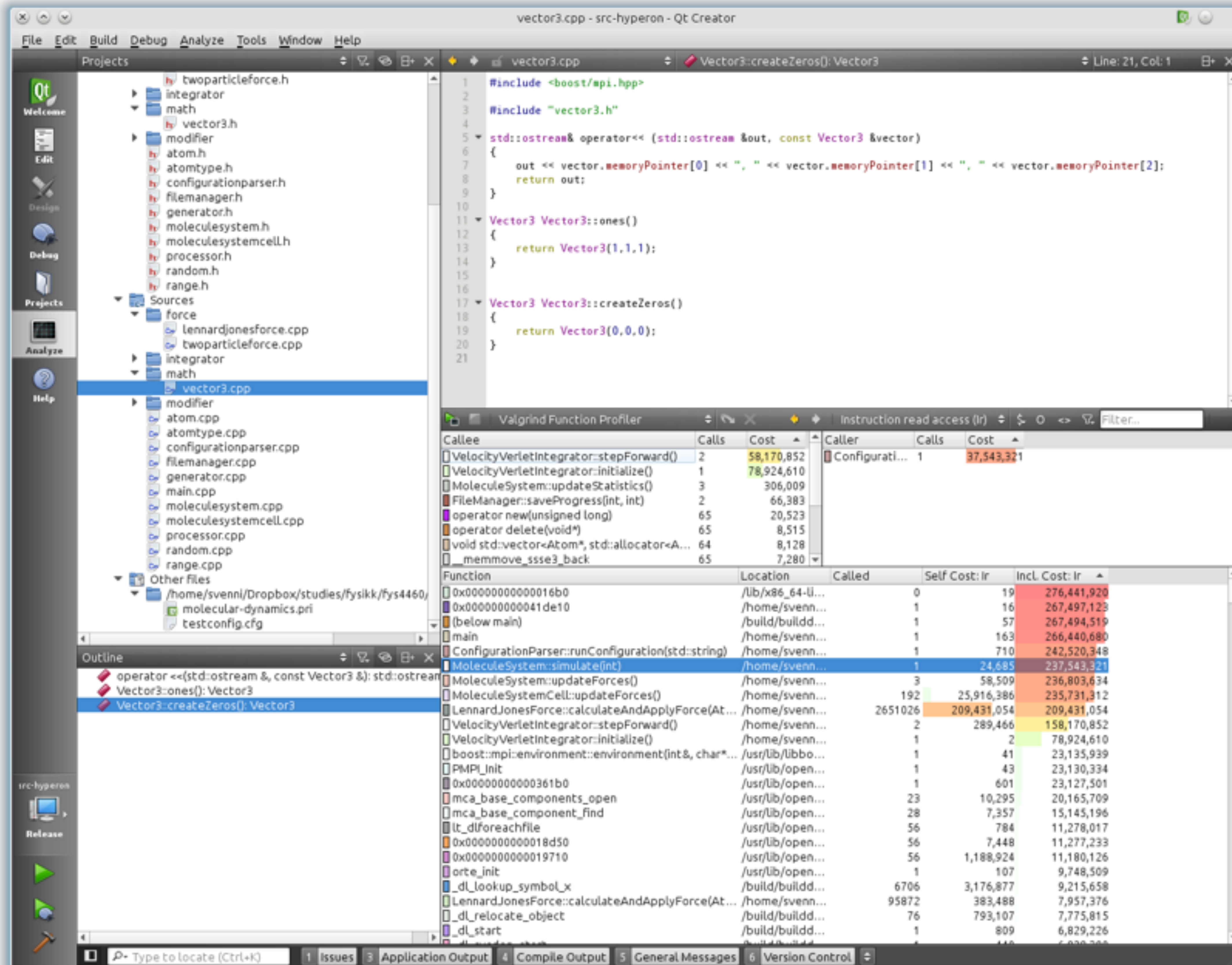
memcheck

```
#include <stdio.h>
#include <stdlib.h>

int main(){
   int *a, b, *c;

   a = (int*)malloc(sizeof(int));
   *a = b;
   printf("*a = %d\n", *a);
   c = (int*)malloc(10*sizeof(int));
   printf("c[11] = %d\n",c[11]);
   return 1;
}
```

```
==20594== 4 bytes in 1 blocks are definitely lost in loss record 1 of 2
==20594== at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==20594==    by 0x400555: main (test_valgrind.c:7)
==20594==
==20594== 40 bytes in 1 blocks are definitely lost in loss record 2 of 2
==20594== at 0x4C2B6CD: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==20594== by 0x400586: main (test_valgrind.c:10)
```



Valgrind On Qt Creator

```
$ valgrind —tool=cachegrind ./test_valgrind
```

Cachegrind

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 100

int main(int argc, char **argv)
{
    int array[SIZE][SIZE] = {0};
    int i,j;
#if 1
    for (i = 0; i < SIZE; ++i) {
        for (j = 0; j < SIZE; ++j) {
            array[i][j] = i + j;
        }
    }
#else
    for (j = 0; j < SIZE; ++j) {
        for (i = 0; i < SIZE; ++i) {
            array[i][j] = i + j;
        }
    }
#endif
    return 0;
}
```

Cache Performance Evaluation Example

$ valgrind —tool=cachegrind ./test_valgrind

Cachegrind

Condition 1

```
==2079== I refs: 219,767
==2079== I1 misses: 614
==2079== L2i misses: 608
==2079== I1 miss rate: 0.27%
==2079== L2i miss rate: 0.27%
==2079==
==2079== D refs: 124,402 (95,613 rd + 28,789 wr)
==2079== D1 misses: 2,041 ( 621 rd + 1,420 wr)
==2079== L2d misses: 1,292 ( 537 rd + 755 wr)
==2079== D1 miss rate: 1.6% ( 0.6% + 4.9% )
==2079== L2d miss rate: 1.0% ( 0.5% + 2.6% )
==2079==
==2079== L2 refs: 2,655 ( 1,235 rd + 1,420 wr)
==2079== L2 misses: 1,900 ( 1,145 rd + 755 wr)
==2079== L2 miss rate: 0.5% ( 0.3% + 2.6% )
```

$ valgrind —tool=cachegrind ./test_valgrind

Cachegrind

Condition 2

```
==2080== I refs: 219,767
==2080== I1 misses: 614
==2080== L2i misses: 608
==2080== I1 miss rate: 0.27%
==2080== L2i miss rate: 0.27%
==2080==
==2080== D refs: 124,402 (95,613 rd + 28,789 wr)
==2080== D1 misses: 1,788 ( 621 rd + 1,167 wr)
==2080== L2d misses: 1,292 ( 537 rd + 755 wr)
==2080== D1 miss rate: 1.4% ( 0.6% + 4.0% )
==2080== L2d miss rate: 1.0% ( 0.5% + 2.6% )
==2080==
==2080== L2 refs: 2,402 ( 1,235 rd + 1,167 wr)
==2080== L2 misses: 1,900 ( 1,145 rd + 755 wr)
==2080== L2 miss rate: 0.5% ( 0.3% + 2.6% )
```

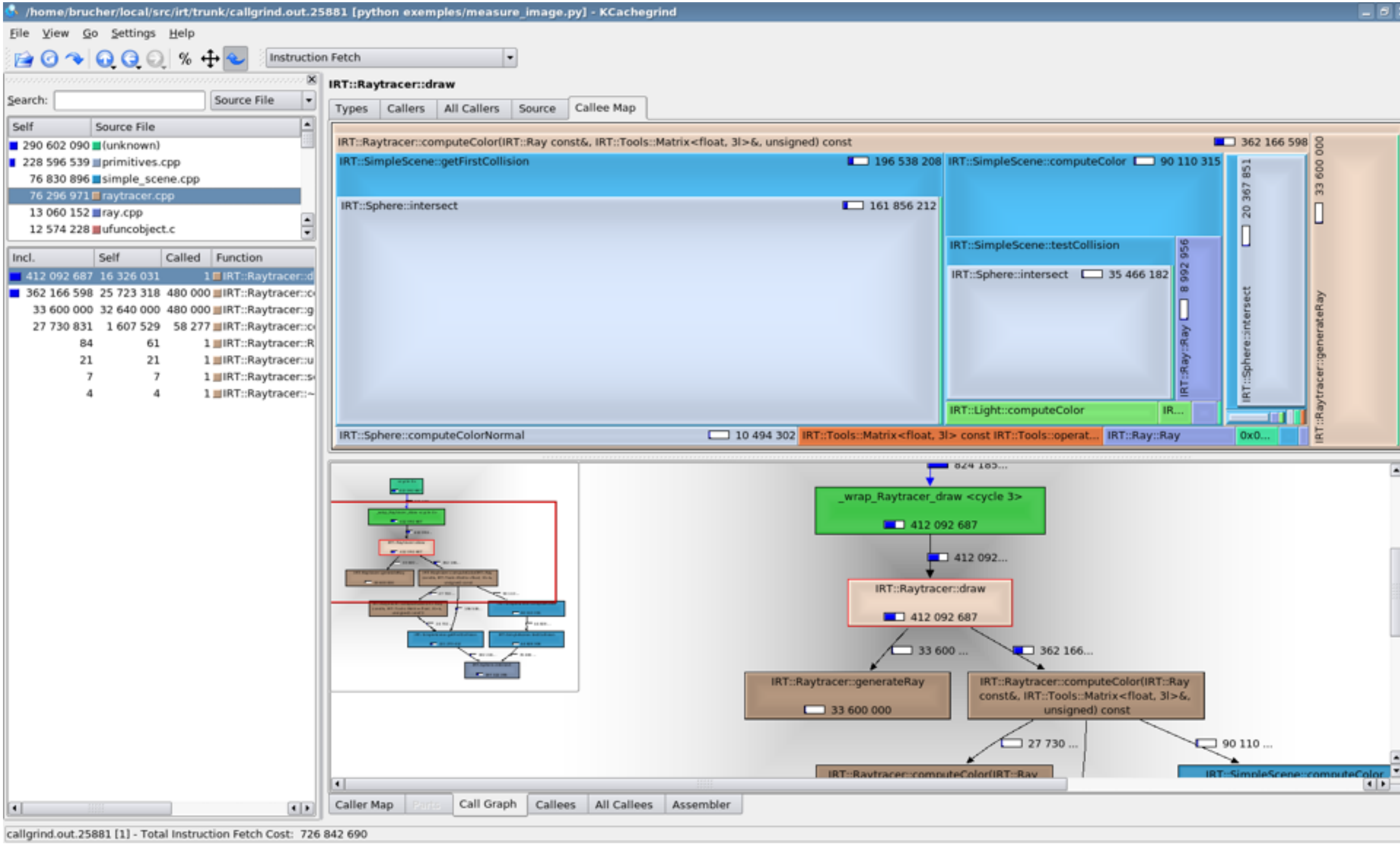
$ valgrind —tool=cachegrind ./test_valgrind

Traditionally, Performance in Condition 1 should better than Condition 2, because of Locality Principle.

What if increase the size from 100 to 1000?
miss rate on condition 1: 1.7%
miss rate on condition 2: 14.5%

Match the Locality Principle again!

Why?



KCacheGrind to Analyze output of Valgrind

Cyber Physical System

- Components will implement features by relying on collaboration with other components that provide part of the required functionality.
- Communicate in the corresponding cyberspace while operating in a physical environment
- Example: (Smart Emergency Response System) https://youtu.be/Yi_dK4iRCA4

Cyber Physical System

Challenges:

- Dynamically creating a configuration
- Achieving a concerted function
- Providing the technological infrastructure

Achieving a concerted function

- Emerging behavior design
- Data sharing
- Functionality sharing
- Collaborative functionality testing

Providing the technological infrastructure

- Design artifact sharing
- Wireless communication
- Hardware resource sharing
- service utilization

Dynamically creating a configuration

Virtual System Integration is a critical need as the physical system in all its potential variants does not become available until runtime.

Need	Challenge	Technology	Impact
Virtual system integration	Proper models in design	Generation of models with necessary detail based on property selection	Confidently design systems as part of a reliable system ensemble
	System-level design and analysis by using models	Connecting, combining, and integrating models represented in different formalisms	
		Efficient simulation models to be used across dynamic and execution semantics	
	Connectivity among models, software, and hardware	Open tool platforms with trusted interfaces for communication across synchronized and coordinated models, software, and hardware devices	

Dynamically creating a configuration

Runtime System Integration,a key implementation closed to CPS. To adapt the system, it is necessary but challenging to maintain the state of the current ensemble of subsystems, to be able to introspect and reason about potential changes, and to find an optimal configuration in the face of available resources.

Need	Challenge	Technology	Impact
Runtime system adaptation	Reasoning and planning adaptation of an ensemble of systems	Introspection of the system state, configuration, and services it makes available	Exploit exogeneous functionality for efficient, economical, and resilient operation
		Maintenance of consistent information and management of inconsistencies regarding the ensemble of systems with sufficient fidelity at runtime	
		Online model calibration	
	Testing with functionality on deployed systems	Environment models to enable surrogate interactions	

Dynamic Adaption

Kevoree model

- open-source dynamic component model at runtime
- enabling the development of re-configurable distributed system
- Support: Java, Android, MiniCloud, FreeBSD, Arduino

Dynamic Adaptation for Microcontrollers

μ-Kevoree

- Types
  plian C
- Asynchronous message passing
  FIFO queue
- Instance scheduler
  Periodic execution
  Triggered execution

Video Demo