Debugging Linux* Systems with GNU* GDB

Agenda

- **Introduction**
- Features
- Debugging on Command Line
- Debugging with Eclipse* IDE
- Documentation
- Summary

Introduction

Why do I need a debug solution?

- Development is a complicated task: You usually apply many **different technologies** throughout a typical developer's day, without always knowing.
- Some **new technologies** (e.g. C++11, vectorization, Intel® Processor Trace, …) might have strange effects. A debugger can help you understand how they work and help validating whether your code's semantic is as expected.
- A debugger helps you to **narrow down problems** caused by deviations from the expected code's behavior.
- It is crucial for a debugger to have **awareness of the underlying processor architecture**. Basic features of the architecture that aid in debugging should also be usable.

A debugger does not only help you finding problems but also shows you how things work!

Introduction

Debug solution from Intel:

- Command line with GNU* GDB
- Eclipse* IDE

How to get it?

- Intel® System Studio:
  https://software.intel.com/en-us/intel-system-studio

For best experience use debug solution from the most recent version!

Introduction

Debug solution from Intel® based on GNU* GDB 7.7:

- Included with Intel® System Studio 2013 and later
- Complements debuggers you might already have

Why to use provided GNU* GDB?

- Capabilities are released back to GNU* community
- Latest GNU* GDB versions in future releases
- Improved C/C++ support thanks to Project Archer and contribution through Intel
- Increased support for Intel® architecture
- Additional debugging capabilities – more later

Introduction

Why different flavors?

- Command line with GNU* GDB:
  - Well known syntax
  - Lightweight: no dependencies
  - Easy setup: no project needs to be created
  - Can be automatized/scripted
- Eclipse* IDE:
  - Comfortable user interface
  - Well known cross-platform IDE
  - Use existing Eclipse* projects
  - Simple integration of the Intel enhanced GNU* GDB
  - Remote System Explorer (RSE) for managing remote systems (targets)

Agenda

- Introduction
- **Features**
- Debugging on Command Line
- Debugging with Eclipse* IDE
- Documentation
- Summary

Features

Overview

Intel's GNU* GDB provides additional extensions that are available on the command line:

- **Data Race Detection (*pdbx*):**
  Detect and locate data races for applications threaded using POSIX* thread (pthread) or OpenMP* models

- **Branch Trace Store (*btrace*):**
  Record branches taken in the execution flow to backtrack easily after events like crashes, signals, exceptions, etc.

- **Pointer Checker:**
  Assist in finding pointer issues if compiled with Intel® C++ Compiler and having Pointer Checker feature enabled

Features

Data Race Detection (PDBX) I

What are data races?

- A data race happens…
  If at least two threads/tasks access the same memory location w/o synchronization and at least one thread/task is writing.

- Example:

```
int a = 1;
int b = 2;
…
int thread1() {
  return a + b;
}
```

```
int thread2() {
  b = 42;
}
```

time

Return value of `thread1()` depends on timing: 3 vs. 43!

Features

Data Race Detection (PDBX) II

What are typical symptoms of data races?

- Data race symptoms:
  - Corrupted results
  - Run-to-run variations
  - Corrupted data ending in a crash
  - Non-deterministic behavior

- Solution is to synchronize concurrent accesses, e.g.:
  - Thread-level ordering (global synchronization)
  - Instruction level ordering/visibility (atomics)
    Note:
    Race free but still not necessarily run-to-run reproducible results!
  - No synchronization: data races might be acceptable

⇨ **GDB data race detection can analyze correctness**

Features

Data Race Detection (PDBX) III

How to detect data races?

- Prepare to detect data races:
  - Only supported with Intel® C++ Compiler:
    Compile with `-debug parallel` (`icc` or `icpc`)
    Only objects compiled with `-debug parallel` are analyzed!
  - Optionally, add debug information via `-g`
- Enable data race detection (PDBX) in debugger:

```
(gdb) pdbx enable
(gdb) c
data race detected
1: write shared, 4 bytes from foo.c:36
3: read shared, 4 bytes from foo.c:40
Breakpoint -11, 0x401515 in L_test_..._21 () at foo.c:36
*var = 42; /* bp.write */
```

Features

Data Race Detection (PDBX) IV

Requires additional library **`libpdbx.so.5`**

- Keeps track of the synchronizations
- Part of Intel® C++ Compiler

Supported parallel programming models:

- OpenMP*
- POSIX* threads

Data race detection can be enabled/disabled at any time

- Only memory access are analyzed within a certain period
- Keeps memory footprint and run-time overhead minimal

There is finer grained control for minimizing overhead and selecting code sections to analyze

⇨ **Filter sets**

Features

Data Race Detection (PDBX) V

More control about what to analyze with filters:

- Add filter to selected filter set

```
(gdb) pdbx filter line foo.c:36
(gdb) pdbx filter code 0x40518..0x40524
(gdb) pdbx filter var shared
(gdb) pdbx filter data 0x60f48..0x60f50
(gdb) pdbx filter reads # read accesses
```

- Ignore events specified by filters (default behavior)

```
(gdb) pdbx fset suppress
```

- Ignore events not specified by filters

```
(gdb) pdbx fset focus
```

exclusive

- Get debug command help

```
(gdb) help pdbx
```

Features

Data Race Detection (PDBX) VI

Use cases for filters:

- Focused debugging, e.g.:
  Debug a single source file and only focus on one specific memory location.

- Limit overhead and control false positives

- Exclude 3rd party code

Features

Data Race Detection (PDBX) VII

Hint:

- Optimized code (symptom):

```
(gdb) run
data race detected
1: write question, 4 bytes from foo.c:36
3: read question, 4 bytes from foo.c:40
Breakpoint -11, 0x401515 in foo () at foo.c:36
 *answer = 42;
(gdb)
```

- Incident has to be analyzed further:
  - Remember: data races are reported on memory objects
  - If symbol name cannot be resolved: only address is printed
- **Recommendation:**
  Unoptimized code (`-O0`) makes it easier to understand
  (due to removed/optimized away temporaries, etc.)

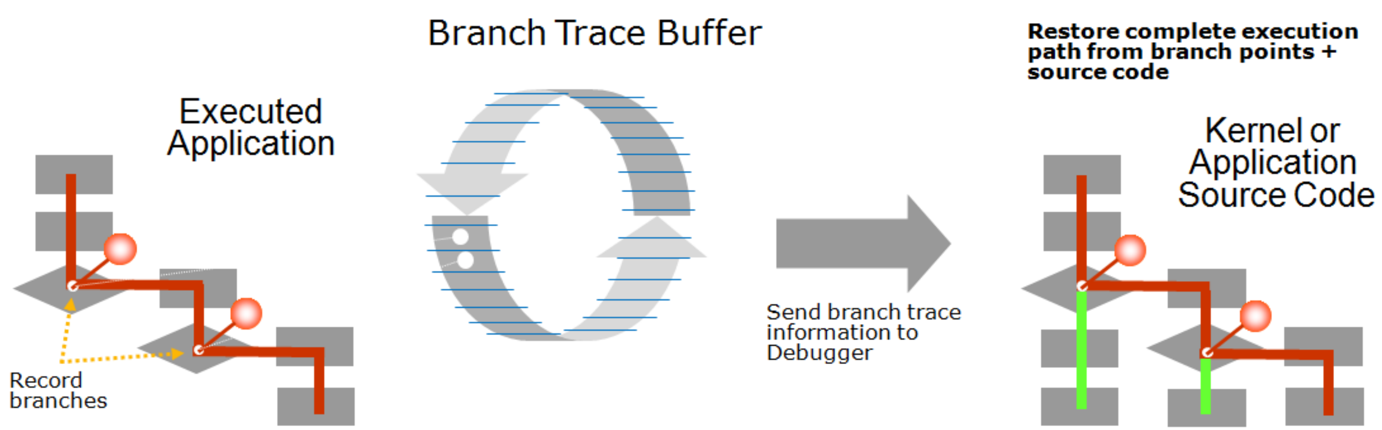
Features

Data Race Detection (PDBX) VIII

One more hint:

- Reported data races appear to be false positives:
  - Not all data races are bad… user intended?
  - OpenMP*: Distinct parallel sections using the same variable (same stack frame) can result in false positives

Features
Branch Trace Store I



- Intel® Atom™ and Intel® Core™ (Haswell and higher) processors supports Branch Trace Store (BTS)
- Allows to unroll execution flow at any point in time

⇨ **Where did things start to go wrong?**

Features

Branch Trace Store II

- Integrated into *process record and replay* subsystem of GNU* GDB 7.6 and later.
- **However:**
  BTS does not allow replaying and reverse execution

- Branch trace per thread (HW & SW)
- Show compact control flow overview
- Show detailed execution trace disassembly (history)

⇨ **Answers: How did I get here?**

Features

Branch Trace Store III

Commands:

- Enable/disable BTS:
  `record btrace/record stop`
  Start/stop recording the branch trace for all threads
- Status of BTS (and *process record and replay* in general):
  `info record`
- Display list of basic blocks:
  `record function-call-history [/l] [/i]`
- Display disassembly of instructions in BTS log:
  `record instruction-history [#-of-instruction]`
- Much more commands; mostly for configuration and displaying!
  See `GDB.pdf` for more information.

Features

Branch Trace Store IV

Useful for:

- Debugging internal state corruption
- Debugging stack corruption (broken backtrace)
- Quick control flow overview

Overhead up to 60x possible!
⇨ Use selective enabling to reduce it

Features

Branch Trace Store – Example I

```
#include <stdio.h>

int *getmem(int i)
{
    // buggy memory pool
    int memory[10];
    return &memory[i];
}

void memzero(int *ptr)
{
    *ptr = 0;
}

void work(int *ptr)
{
    memzero(ptr);
    // do some work here
}
```

```
int main(int argc, char **argv)
{
    int i;
    int *ptr;

    for(i = 0; i < 10; i++) {
        printf("%d\n", i);
        ptr = getmem(i);
        work(ptr);
    }
    return 0;
}
```

Source file: **segv.c**

Features

Branch Trace Store – Example II

Compiling and executing the example:

```
$ icpc segv.c -o segv -g && ./segv
0
1
2
3
4
Segmentation fault (core dumped)
```

Something went wrong here. Let's use the debugger to learn why...

Features

Branch Trace Store – Example III

First run with debugger:

```
$ gdb-ia segv
Reading symbols from segv...done.
(gdb) r
Starting program: segv
0
1
2
3
4

Catchpoint -2 (signal SIGSEGV), work (
    ptr=<error reading variable: Cannot access memory at address
0x7ffeffffff0>)
    at segv.c:18
18 }
...
```

7/16/2014

Features

Branch Trace Store – Example IV

```
...
(gdb) bt
Python Exception <class 'gdb.MemoryError'> Cannot access memory at
address 0x7fff00000008:
#0  work (ptr=<error reading variable: Cannot access memory at
address 0x7ffeffffff0>)
    at segv.c:18
Cannot access memory at address 0x7fff00000008
```

The stack trace cannot be printed. We have no idea what caused the SEGV!

Features

Branch Trace Store – Example V

Second run with BTS:

```
$ gdb-ia segv
Reading symbols from segv...done.
(gdb) b main
Breakpoint 1 at 0x400613: file segv.c, line 25.
(gdb) r
Starting program: segv

Breakpoint 1, main (argc=1, argv=0x7fffffffddf8) at segv.c:25
25      for(i = 0; i < 10; i++) {
(gdb) record btrace
...
```

Features

Branch Trace Store – Example VI

```
...
(gdb) c
Continuing.
0
1
2
3
4

Catchpoint -2 (signal SIGSEGV), work (
    ptr=<error reading variable: Cannot access memory at address
0x7ffeffffff0>)
    at segv.c:18
18 }
(gdb) info record
Active record target: record-btrace
Recorded 1164 instructions in 60 functions for thread 1 (process
13556).
...
```

Features

Branch Trace Store – Example VII

```
...
(gdb) bt
#0  work (ptr=<error reading variable: Cannot access memory at
address 0x7ffeffffff0>)
    at segv.c:18
Cannot access memory at address 0x7fff00000008
(gdb) record function-call-history /l /i
50  1091            free@plt
51  1092-1098           free
52  1099-1124           vfprintf
53  1125-1126           printf
54  1127-1130   segv.c:26-27    main
55  1131-1141   segv.c:4-6  getmem(int)
56  1142-1147   segv.c:27-28    main
57  1148-1154   segv.c:15-16    work(int*)
58  1155-1162   segv.c:10-12    memzero(int*)
59  1163     segv.c:18   work(int*)
```

The last executed function was **`work(…)`**. It's return (line 18) triggered the SEGV!

Features

Pointer Checker

Debugging:

```
int arr[N];
int last(int* arr) {
      return arr[N - 1];
}
void bug() {
      int i = last(arr + 1);
}
```

- Compile application with **-g** & **-check-pointers=[rw|write]**
- It extends pointer with bounds information and checks bounds before each access
- New command: **info sbounds <ptr>**

```
(gdb) c
Continuing.
Upper bound violation.
With bounds {0xec8, 0xeef} accessing at: 0xef0 and size: 4.
Temporary breakpoint -13, 0x40090b in last (arr=0xecc <arr+4>) at
bounds.c:3
4         return arr[N - 1];
(gdb) info sbounds arr
{lbound = 0xec8, ubound = 0xeef}: ptr. value = 0xecc, size = 40
```

Agenda

- Introduction
- Features
- **Debugging on Command Line**
- Debugging with Eclipse* IDE
- Documentation
- Summary

Debugging on Command Line

Remote Debugging I

1. Select **gdbserver** for the target:
   - Linux host:
     `<install-dir>/system_studio_2015.x.y/targets/`
     `<arch>/<target>/bin/gdbserver`
   - Windows host:
     `<install-dir>\System Studio 2015.x.y\targets\`
     `<arch>\<target>\bin\gdbserver`
   - **<arch>** can be: **ia32**, **intel64** or **Quark**
   - **<target>** is one of the supported target systems, e.g.: **Galileo**, **Yocto1.5**, **WindRiverLinux5**, **TizenIVI**, …

2. Copy selected **gdbserver** to the target

Debugging on Command Line

Remote Debugging II

3. Start **`gdbserver`** on the target:
   - Loading application called **`<application>`**:
     ```
     $ gdbserver localhost:2000 <application>
     Process program created; pid = …
     Listening on port 2000
     ```
   - Attach to running application with PID **`<pid>`**:
     ```
     $ gdbserver localhost:2000 --attach <pid>
     Process program created; pid = …
     Listening on port 2000
     ```

   Note:
   Application has to be launched externally before. To avoid runaway, use:
   ```
   …
   volatile int lock_loop = 1;
   while(lock_loop) { lock_loop = 1; } // or: sleep(1)
   …
   ```
   When attached set **`lock_loop`** manually **`0`** to continue.

Debugging on Command Line

Remote Debugging III

4. Start GDB on host:

   - Linux host:

     ```
     $ source <install-dir>/system_studio_2015.x.y/bin/
                    compilervars.[sh|csh] [ia32|ia32_intel64]
     $ gdb-ia <application>
     ```

   - Windows host:

     ```
     $ call <install-dir>\System Studio 2015.x.y\bin\
                    compilervars.bat [ia32|ia32_intel64]
     $ gdb-ia.bat <application>
     ```

5. Connect GDB to **`gdbserver`** on target:

   ```
   (gdb) target remote <target>:2000
   Remote debugging using <target>:2000
   ```

   `<target>` is network name/IP of target system.

Debugging on Command Line

Remote Debugging with Multi-Mode

Alternative usage: Multi-Mode

1. Start **gdbserver** on the target:
   ```
   $ gdbserver --multi localhost:2000
   Listening on port 2000
   ```

2. Start GDB on host:
   ```
   $ gdb-ia[.bat]
   (gdb) target extended-remote <target>:2000
   Remote debugging using <target>:2000
   (gdb) file <application>
   ```

   - Attach:
     ```
     (gdb) attach <pid>
     ```
     <application> is host path, **<pid>** is PID on the target

   - Load & execute:
     ```
     (gdb) set remote exec-file <application_target>
     ```
     <application> is host path, **<application_target>** is target path

Debugging on Command Line

Additional Hints for Remote Debugging

- Sysroot:
  `(gdb) set sysroot <path>`
  Set the path to the system root. `<path>` is the location of system libraries for target on host.

- Search path:
  `(gdb) set solib-search-path <path>`
  `<path>` is path to additional libraries from target to add to the search path on host. Alternatively `$LD_LIBRARY_PATH` can be set on target before launching GDBServer.

- Path substitution:
  `(gdb) set substitute-path <from> <to>`
  Change paths from `<from>` to `<to>`. You can relocate a whole source (sub-)tree with that.

Debugging is no different than known from non-Intel GNU* GDB versions!

See `GDB.pdf` and the Intel® System Studio Product Guide for more information.

Debugging on Command Line

Get Started with PDBX I

- PDBX has some pre-requisites that must be fulfilled for proper operation
- Use **`pdbx check`** command to see whether PDBX is working:

1. First step:
   ```
   (gdb) pdbx check
   checking inferior...failed.
   ```
   Solution:
   Start a remote application (inferior) and hit some breakpoint (e.g. **`b main`** & **`run`**)

2. Second step:
   ```
   (gdb) pdbx check
   checking inferior...passed.
   checking libpdbx...failed.
   ```
   Solution:
   Use **set solib-search-path <lib_paths>** to provide the path of **`libpdbx.so.5`** on the host.

Debugging on Command Line

Get Started with PDBX II

3. Third step:

```
(gdb) pdbx check
checking inferior...passed.
checking libpdbx...passed.
checking environment...failed.
```

Solution:

Set additional environment variables on the target for OpenMP*. Those need to be set with starting GDBServer (similar to setting **$LD_LIBRARY_PATH**).

- **$INTEL_LIBITTNOTIFY32=""**
- **$INTEL_LIBITTNOTIFY64=""**
- **$INTEL_ITTNOTIFY_GROUPS=sync**

Agenda

- Introduction
- Features
- Debugging on Command Line
- **Debugging with Eclipse* IDE**
- Documentation
- Summary

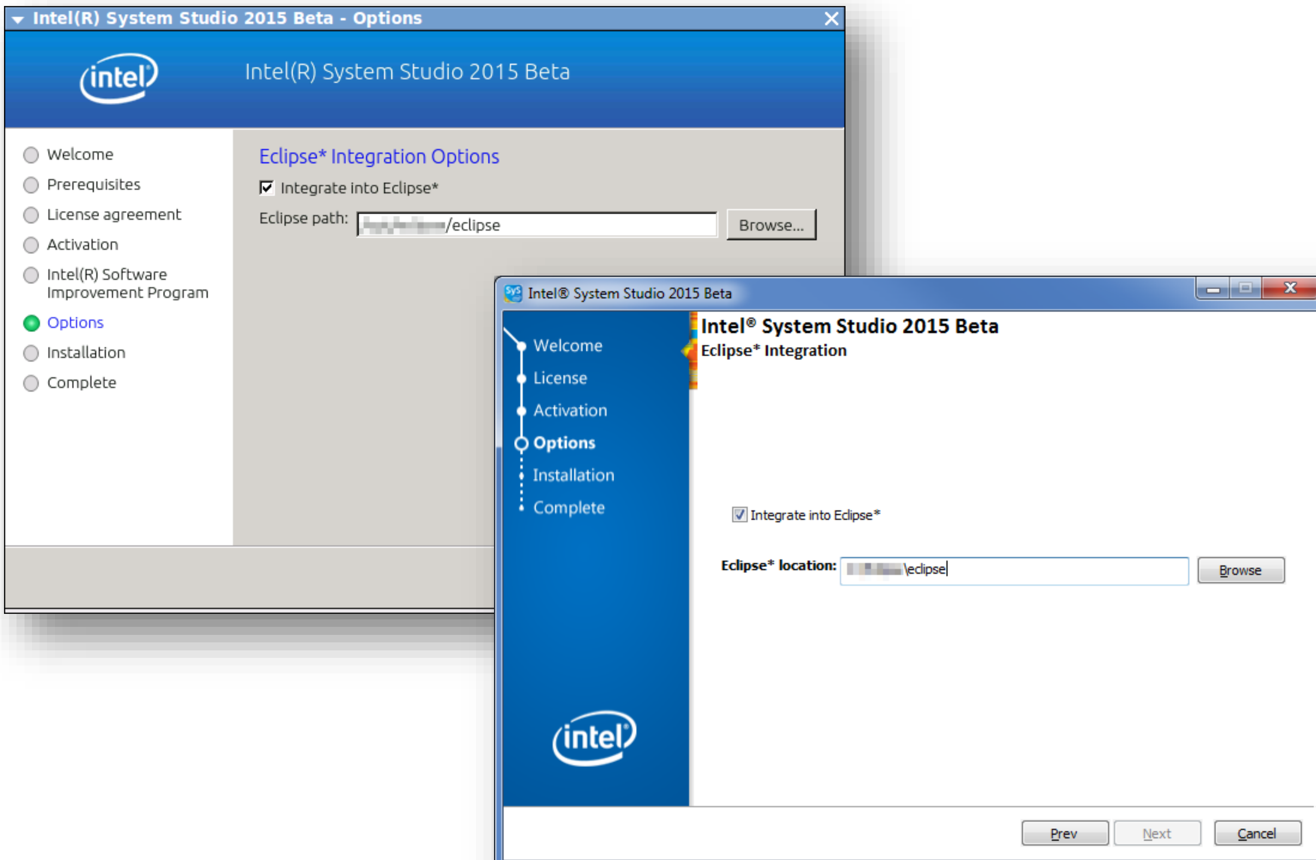
Debugging with Eclipse* IDE
Pre-Requisites I

- Eclipse* IDE:
  - 4.3 with Eclipse C/C++ Development Tools (CDT) 8.1 or later
  - 4.2 with Eclipse C/C++ Development Tools (CDT) 8.1 or later

  We recommend: *Eclipse IDE for C/C++ Developers* (4.3)
- Java* Runtime Environment (JRE) 6.0 or later
- Remote System Explorer (RSE) plugin (optional but recommended)
- Configure RSE to establish connection with the target hardware. Please refer to the RSE documentation for more information: http://www.eclipse.org/tm/tutorial/index.php
- Source/call **`compilervars.[sh|csh|bat]`** before starting Eclipse* IDE (in same environment)

Debugging with Eclipse* IDE
Pre-Requisites II

Integrate into existing Eclipse* IDE during installation:



Debugging with Eclipse* IDE

Remote Debugging I

1. Select **gdbserver** for the target:
   - Linux host:
     <install-dir>/system_studio_2015.x.y/targets/<arch>/<target>/bin/gdbserver
   - Windows host:
     <install-dir>\System Studio 2015.x.y\targets\<arch>\<target>\bin\gdbserver
   - **<arch>** can be: **ia32**, **intel64** or **Quark**
   - **<target>** is one of the supported target systems, e.g.: **Galileo**, **Yocto1.5**, **WindRiverLinux5**, **TizenIVI**, …
2. Copy selected **gdbserver** to the target

Debugging with Eclipse* IDE

Remote Debugging II

3. Start `gdbserver` on the target:

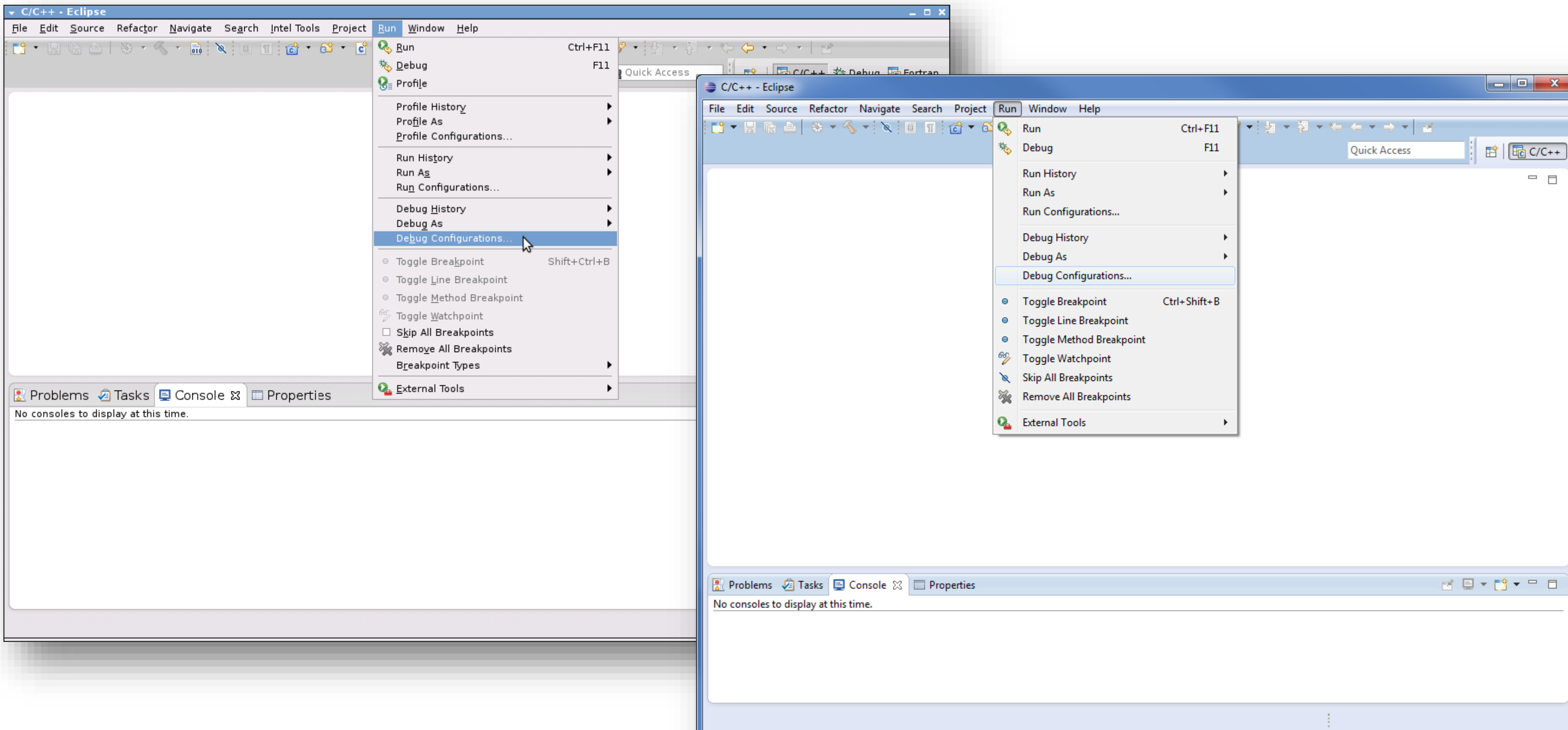
   - Loading application called `<application>`:
     ```
     $ gdbserver localhost:2000 <application>
     Process program created; pid = …
     Listening on port 2000
     ```
   - Attach to running application with PID `<pid>`:
     ```
     $ gdbserver localhost:2000 --attach <pid>
     Process program created; pid = …
     Listening on port 2000
     ```

   Note:
   Application has to be launched externally before. To avoid runaway, use:
   ```
   …
   volatile int lock_loop = 1;
   while(lock_loop) { lock_loop = 1; } // or: sleep(1)
   …
   ```
   When attached set `lock_loop` manually `0` to continue.

Debugging with Eclipse* IDE

Remote Debugging III

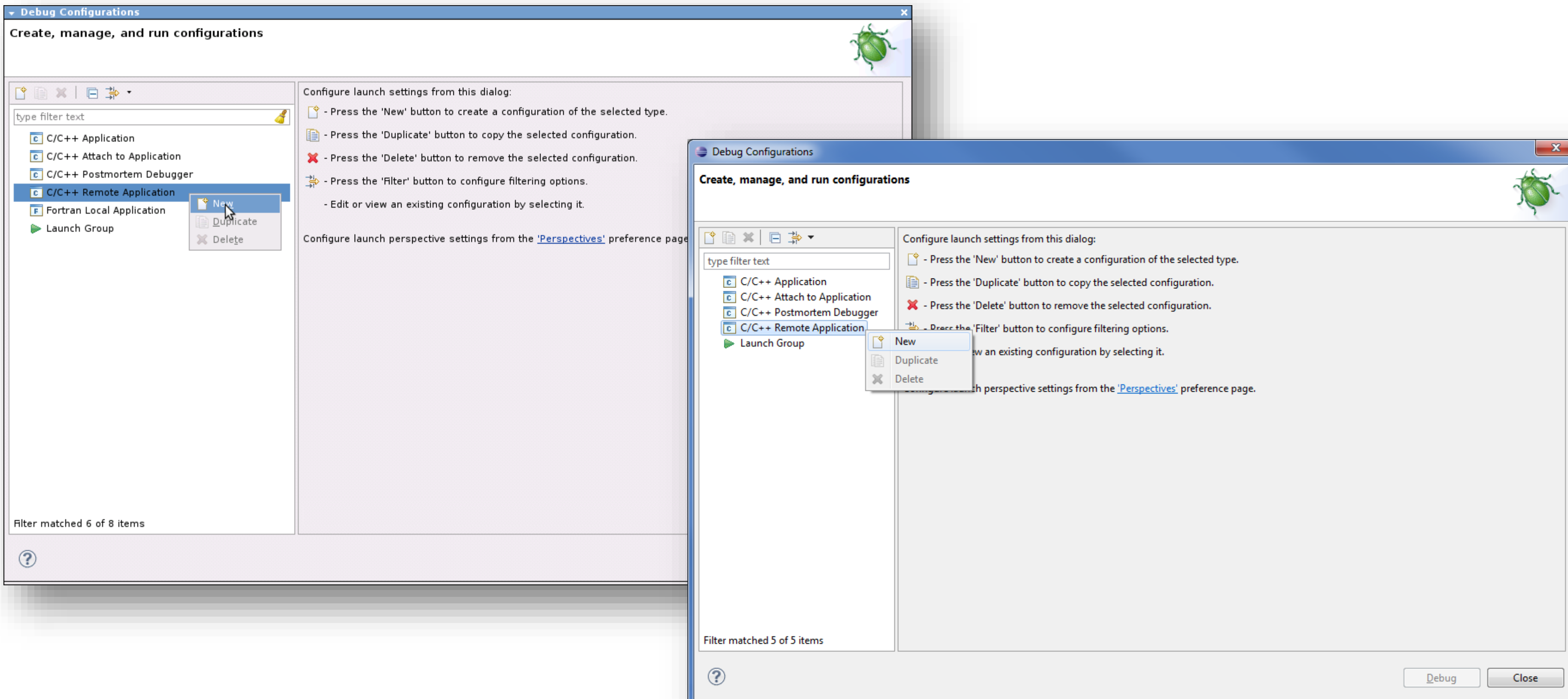
4. Configure Eclipse* IDE to use the provided GDB:
   a. In Eclipse* IDE click on “Run>Debug Configurations…”



Debugging with Eclipse* IDE

Remote Debugging IV

b. Create a new or update an existing “C/C++ Remote Application” or “C/C++ Attach to Application” configuration.

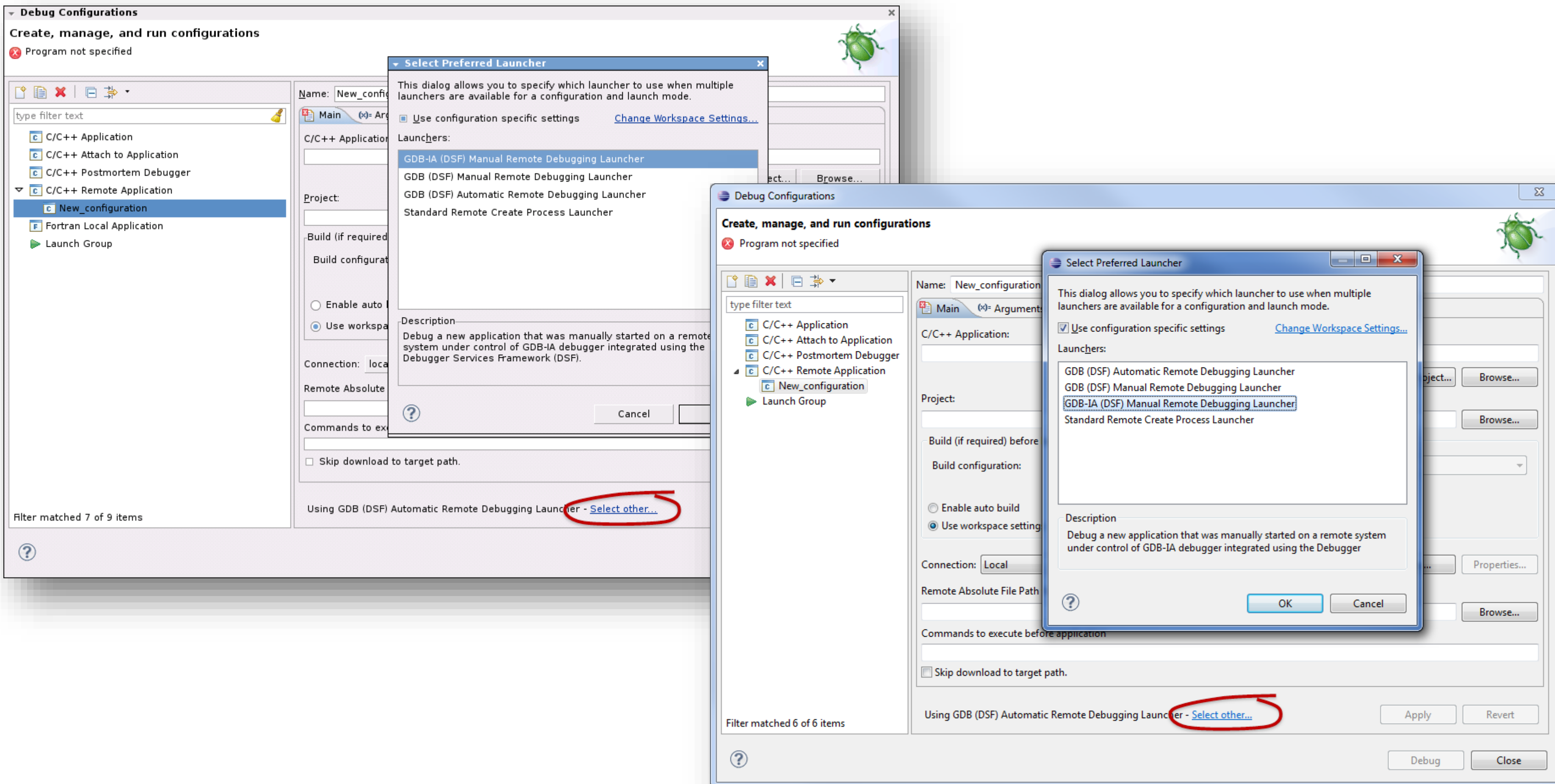


7/16/2014

Debugging with Eclipse* IDE

Remote Debugging V

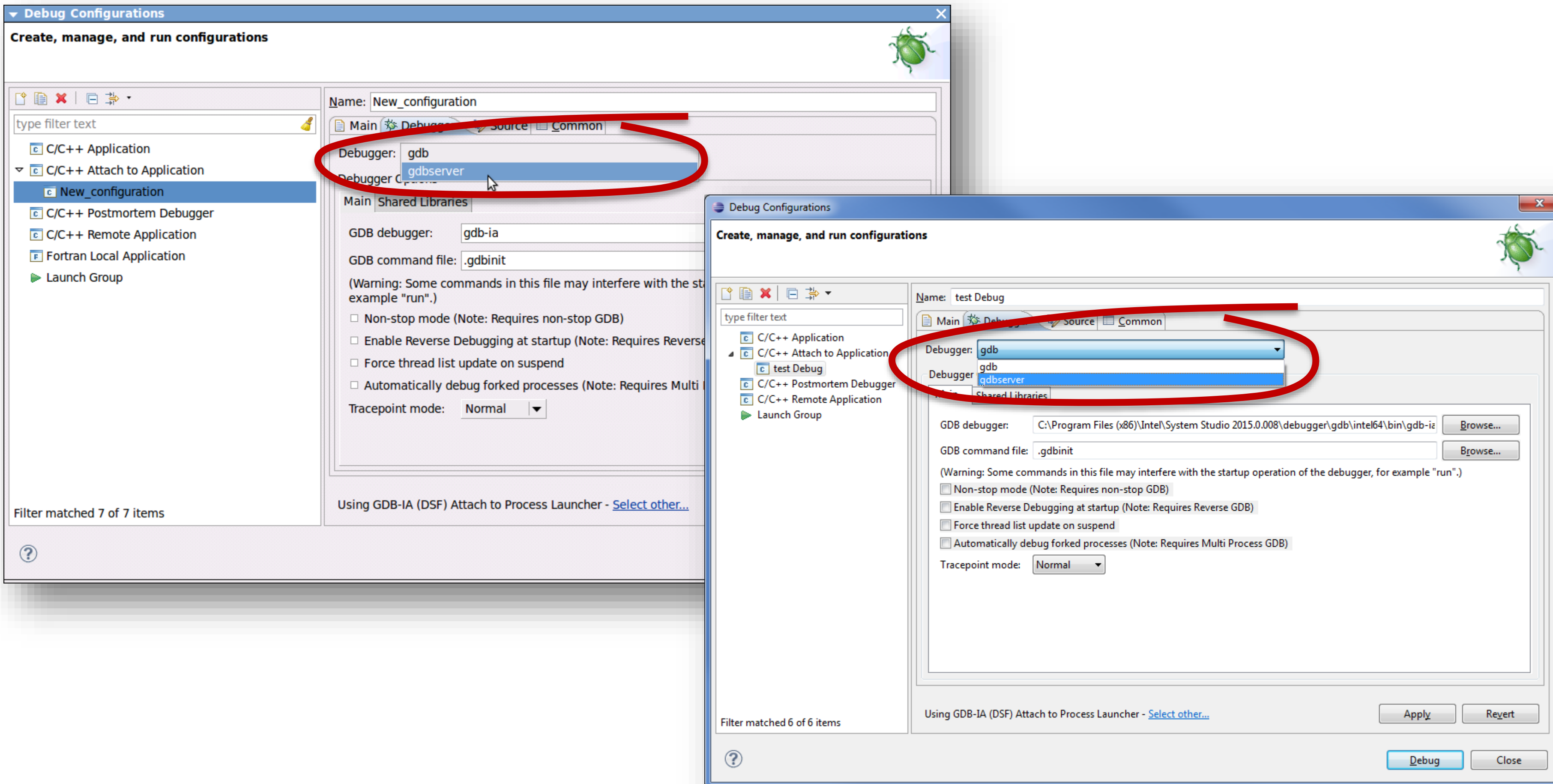
c. Select the launcher: “GDB-IA (DSF)” via link Select other…:



Debugging with Eclipse* IDE

Remote Debugging VI

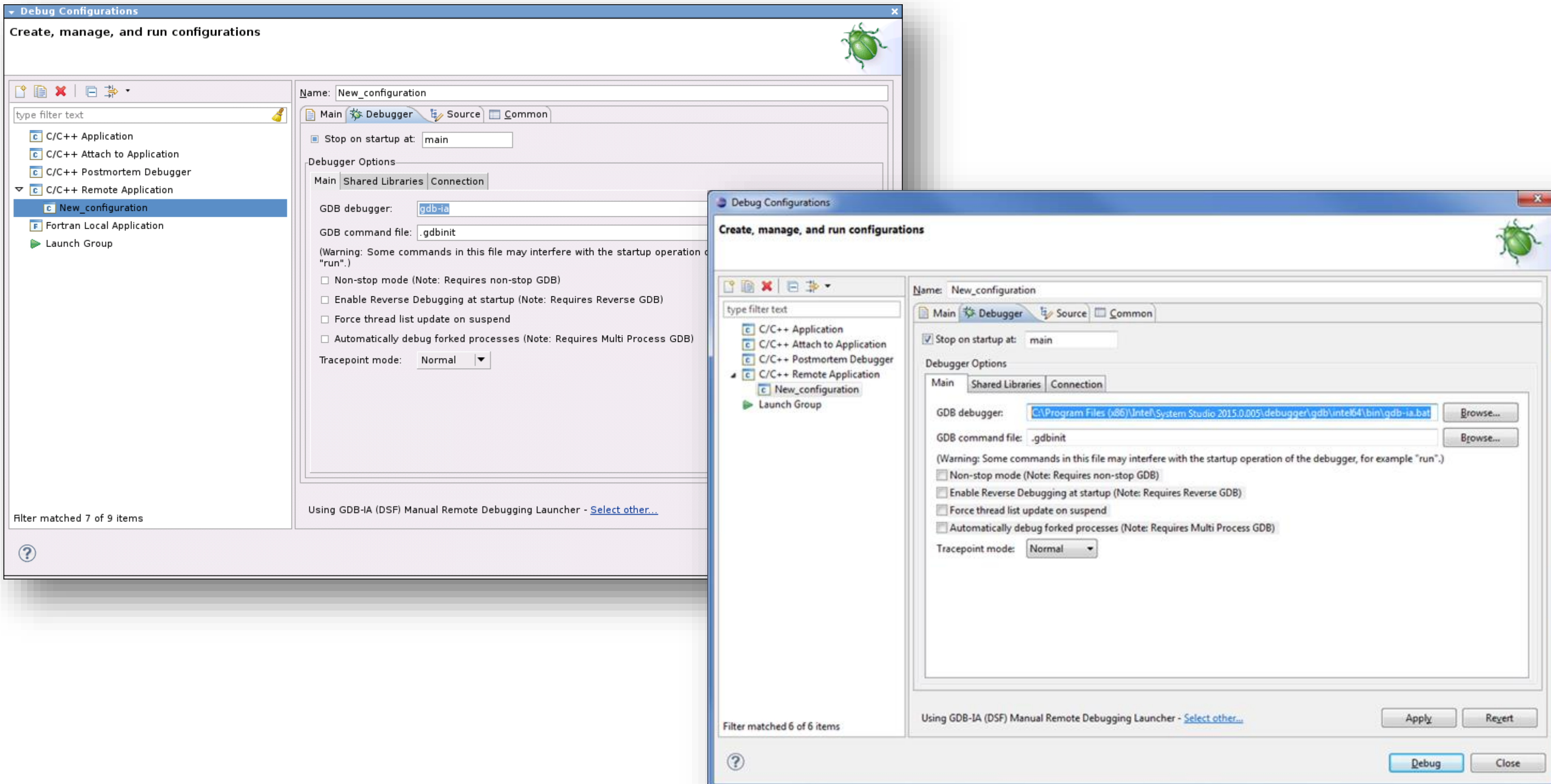
d. In case of attaching, select “gdbserver” as the debugger:



Debugging with Eclipse* IDE

Remote Debugging VII

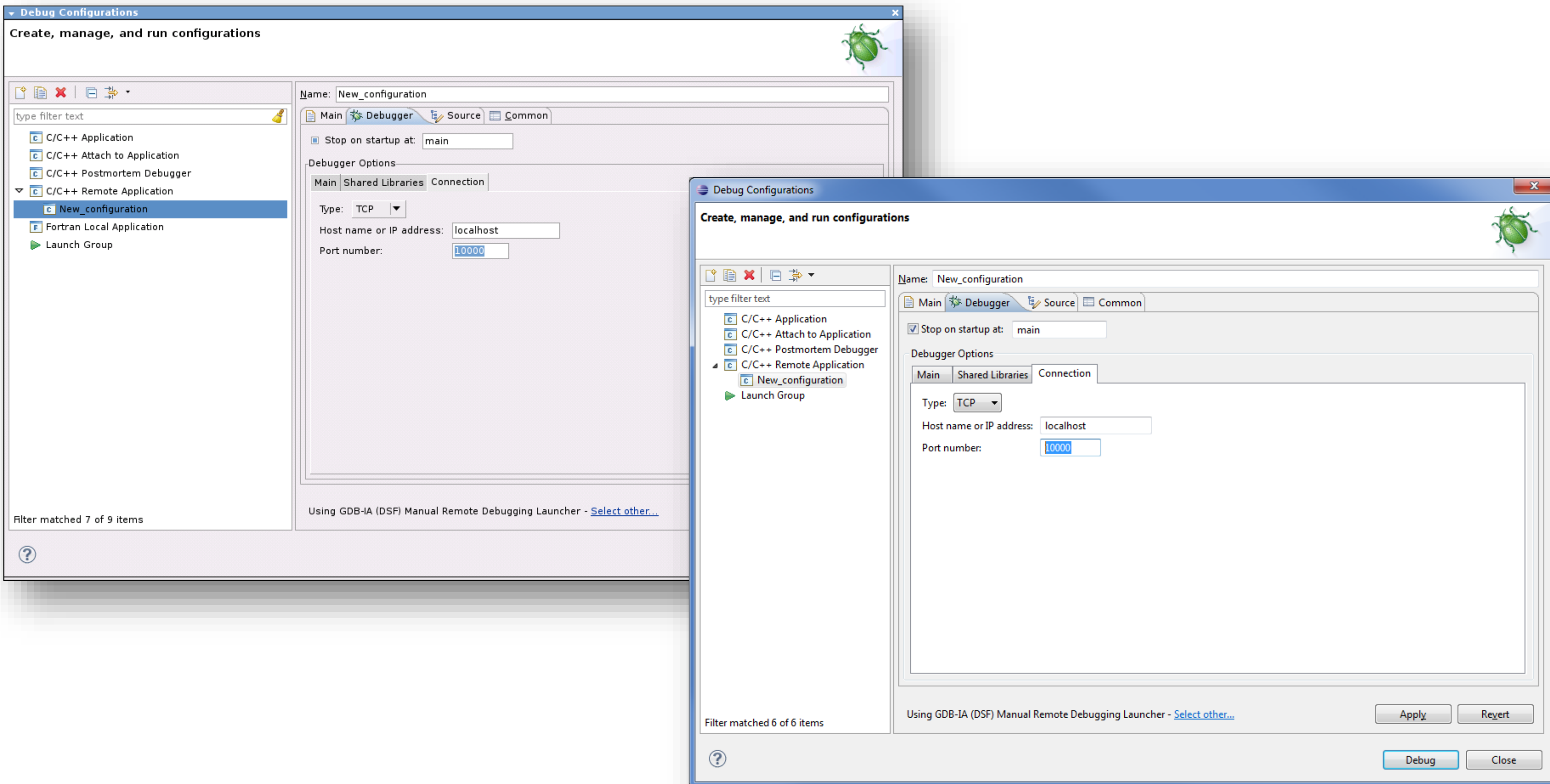
e. The launcher automatically sets the provided GDB:



Debugging with Eclipse* IDE

Remote Debugging VIII

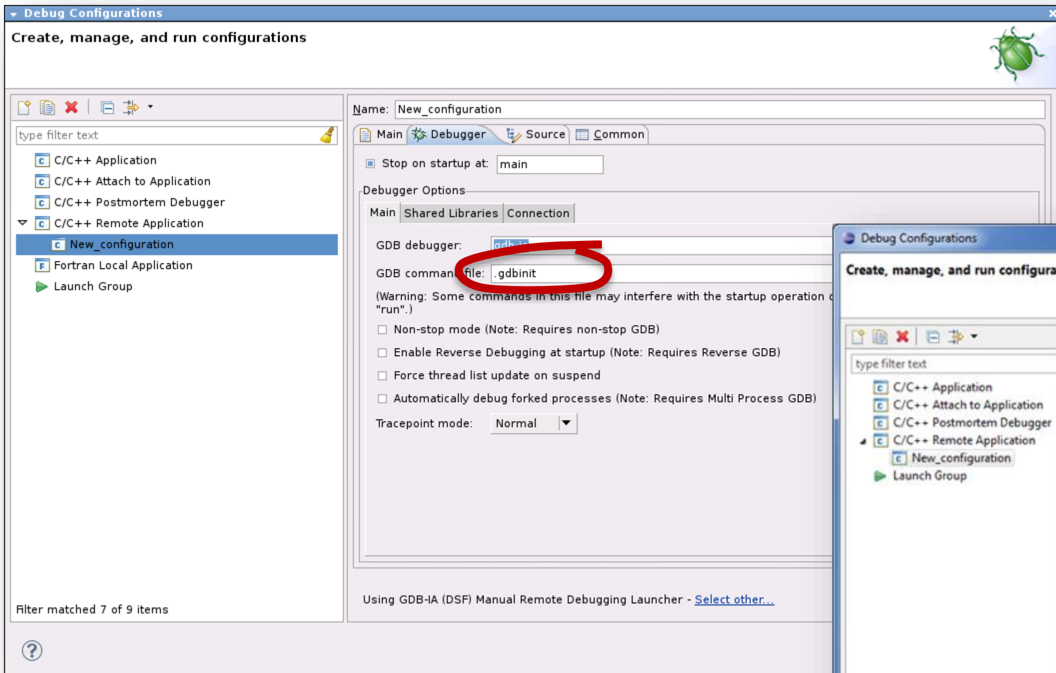
f. Specify the target IP address and port number for **gdbserver**:

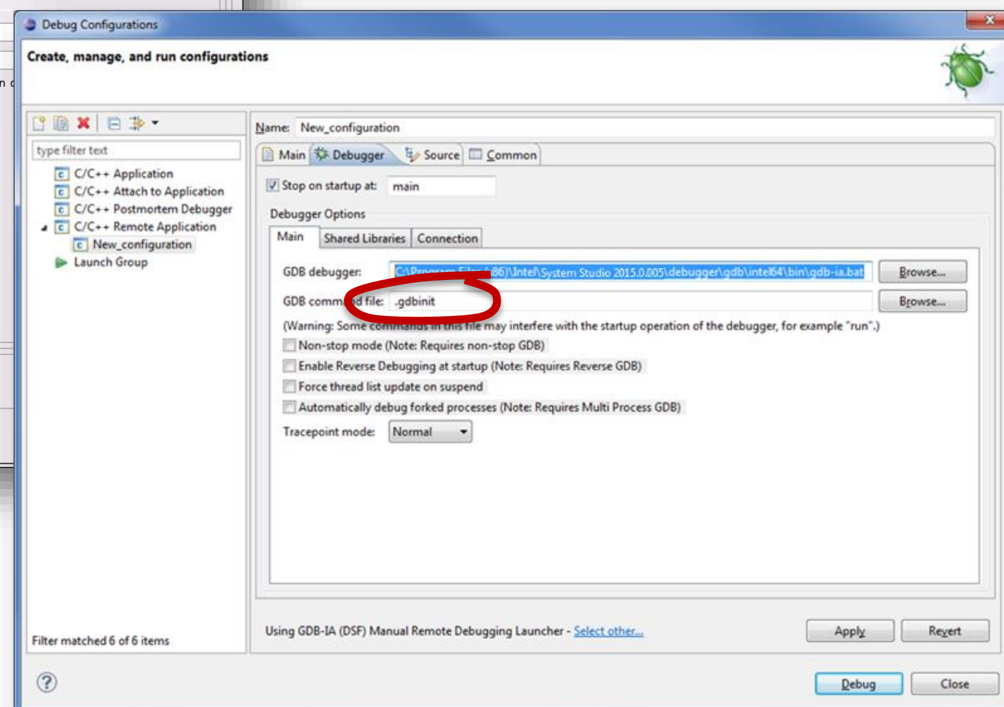


7/16/2014

Debugging with Eclipse* IDE
Additional Hints for Remote Debugging I

Create and configure `.gdbinit`:





7/16/2014

Debugging with Eclipse* IDE

Additional Hints for Remote Debugging II

- Sysroot:
  `(gdb) set sysroot <path>`
  Set the path to the system root. **`<path>`** is the location of system libraries for target on host.
- Search path:
  `(gdb) set solib-search-path <path>`
  `<path>` is path to additional libraries from target to add to the search path on host. Alternatively **`$LD_LIBRARY_PATH`** can be set on target before launching GDBServer.
- Path substitution:
  `(gdb) set substitute-path <from> <to>`
  Change paths from **`<from>`** to **`<to>`**. You can relocate a whole source (sub-)tree with that.

Debugging is no different than known from non-Intel GNU* GDB versions!

See **`GDB.pdf`** and the Intel® System Studio Product Guide for more information.

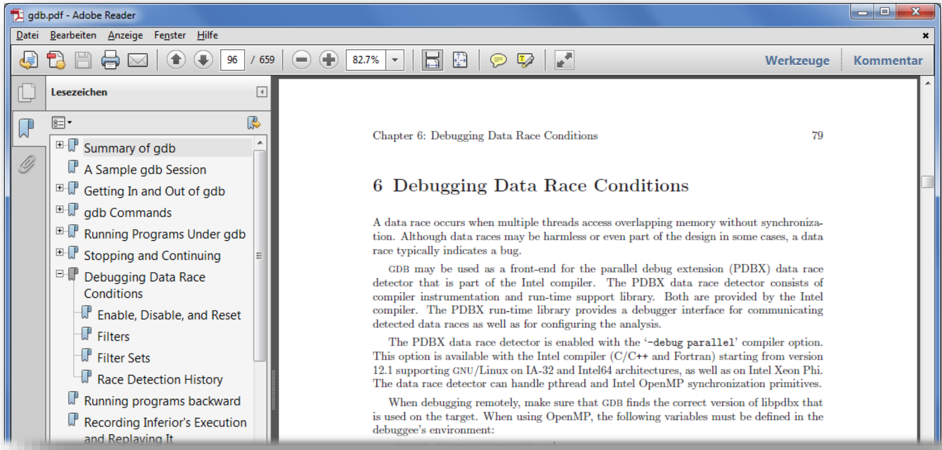
Agenda

- Introduction
- Features
- Debugging on Command Line
- Debugging with Eclipse* IDE
- **Documentation**
- Summary

Documentation

Documentation can be found here:

<system_studio_root>/Documentation/en_US/debugger/gdb/GDB.pdf



Release Notes:

https://software.intel.com/en-us/articles/intel-system-studio-release-notes

...or here:

<system_studio_root>/Documentation/en_US/debugger/gdb/GDB_Release_notes.pdf

Agenda

- Introduction
- Features
- Debugging on Command Line
- Debugging with Eclipse* IDE
- Documentation
- **Summary**

Summary

- Intel's GNU* GDB
  - Well known Syntax
  - Seamless integration of Intel extensions
  - Latest Intel® architecture support
  - Contributing to community
  - Lightweight remote debugging of systems

- Eclipse* IDE
  - Can use Intel's GNU* GDB (with some of its features)
  - One of the most famous IDEs on Linux*
  - Support of both C/C++ and Fortran
  - Remote debugging capabilities

Legal Disclaimer & Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804