Institut of Parallel and Distributed Systems
Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart
Germany

Diplomarbeit Nr. 3433

# Development of a Graphical Numerical Accuracy Debugger based on an FPGA Computing System

Kailai Wang

| | |
|---|---|
| **Course of Study:** | Information Technology |
| **Examiner:** | Prof. Dr. Sven Simon |
| **Supervisor:** | M.sc. Wenbin Li |
| **Commenced:** | 18.June 2012 |
| **Completed:** | 18.December 2012 |
| **CR-Classification:** | B.2, D.2.5, D.3.4, G.1.0 |

# Abstract

In scientific computing, the number of floating point operations are increasing along with the higher performance of computers, as well as the larger problem size. Due to the finite representation of real numbers in computers, the calculated results are rounded into the representative numbers, which results in round-off errors. The round-off errors might be propagated as the program runs longer and in the end leads to an unreliable result.

Discrete Stochastic Arithmetic (DSA) provides a method to evaluate the accuracy of computed results and detect numerical instabilities during execution of the program. The DSA has been implemented on an FPGA-based hardware system. The FPGA-based hardware system has N parallel processing blocks so that it can run the same piece of code N times in parallel in different round-off error propagations, which is required by DSA.

In this thesis, based on this hardware architecture, a graphical numerical accuracy debugger is developed. Using this graphical numerical accuracy debugger, the user can debug same piece of code in both PowerPC processors synchronously, without any modification to source codes.

In order to implement the proposed debugging flow, a script has been written to substitute the original underlying debugging engine of SDK. Within the script, a series of functionalities are achieved: GDB input commands catching/forwarding, process calling, GDB output messages catching/forwarding etc. Moreover, with the substitution, it’s able to collect results from all processing blocks and then the number of significant bits can be calculated and presented to users.

# Acknowledgements

I would like to thank my supervisor Mr. Wenbin Li for his kindly help, as well as the support and advices during my entire thesis working period all-long. He always shows a very patient, respectful and warmhearted attitude not only to me, but also to all other colleagues. With the help of him, I'm able to improve my thoughts in a more technical way and have a better understanding on my topic and related knowledge fields.

I'm also very grateful to Prof. Dr. Sven Simon for providing me such a chance to work on this topic, which gives me an opportunity to get my knowledge well applied and practiced.

I'm thankful for the department of Parallel Systems to provide me a comfortable and friendly working environment, together with all these kind and outgoing colleagues.

Lastly, I would take this chance to thank my family for all their love, encouragement and support all the time.

# Contents

## List of Figures

## List of Listings

# 1 Introduction

## 1.1 Background Knowledge

With the increase of computers‘ speed and performance nowadays, the number of the arithmetic operations, especially floating point operations in scientific computations are significantly increased. However, due to the fact that only a finite number of bits in computer can be used to store floating point numbers, round-off operations are needed to fit the real numbers into the finite representation, which results in a *round-off error* against the actual numbers. As more and more floating point operations are performed in a sequence, the error could be propagated, and in the end, at some point, leads to a result which totally differs from the expected one, which is also known as *numerical instability*.

In order to control this round-off error, several methods are developed, such as *interval arithmetic* (IA), *variable-precision arithmetic* (VPA), *discrete stochastic arithmetic* (DSA) etc [1].

IA provides two values for each result, and the exact result is guaranteed to be between those two values [16], and the length of interval between the two values are considered to be the accuracy of the result. However, extra effort, for instance, change of rounding mode after each floating point operation, has to be performed, which dramatically lowers the computational efficiency. In addition, with the increasement of the problem size, the estimation of the numerical accuracy bases on IA is turned out to be very pessimistic and even fails to give results or any useful information for medium-to-large-size problem case [15].

While VPA allows the precision of floating point arithmetic used in the computations to be variable, depending on the problem to be solved and the required accuracy of the results [2]. However, VPA has the main advantage that it is too slow compared to native floating point operations. With the increase of specified precision, the time which the computations cost will also increase dramatically.

DSA, which is much faster than VPA, has meanwhile the advantage over IA that the estimation of numerical accuracy is significantly tighter and independent of problem size [15]. Therefore, DSA is chosen in this thesis as the basis of arithmetic for discussion.

The basic idea of DSA is explained as follows:

1. Run the same piece of code N times independently and synchronously, with random-rounding [4] applied after each floating point operation.
2. With the N results gathered from N runs after each floating point operation, the accuracy (with respect to significant digits) is calculated based on a pre-developped formula and therefore numerical instability can be detected.

## 1.2 Motivation

The round-off error controlling methods mentioned above can be implemented in a either software or hardware way [2][3][4]. As for DSA, there're are also different kinds of implementations:

- A software implementation: CADNA library developped by Labortoire d'informatique de Paris 6 (LIP6) in University Pierre & Marie Curie and CNRS (UMR 7606) [5][6].
- A hardware implementation by R. Avot-Chotin and H. Mehrez [1].

In this thesis, an FPGA-based computing system with two parallel processing blocks is served as the hardware platform support, which is based on a hardware architecture with DSA support, proposed by Wenbin Li in [15].

However, with this FPGA-based computing system, as well as the accompanying Xilinx Tools (XPS,SDK, XMD, etc), while in SDK‘s graphical debugging interface, it is impossible to debug the same piece of code in C-statement level simultaneously and synchronously in both PowerPC processors,  which is required in DSA (running the same piece of code N times, with N = 2). In addition, the precision (number of significant bits) of certain variable cannot be displayed directly to the user. In this case, the user cannot have a clear and convenient view about how accurate the result is.

Thus, a graphical numerical accuracy debugger should be developed and implemented to fulfill the following goals:

1. Debug the same piece of code in both PowerPC processors simultaneously, without any modification to source codes.
2. Gather the value of variables from both processors while the debugging process is in background execution.
3. Calculate the numerical accuracy and display both the accuracy information and the computed result to users.

## 1.3 Hardware Platform and Software Tools

### 1.3.1 Hardware Devices

The following hardware devices are used in this thesis:

- Xilinx Virtex-5 FXT ML510 FPGA board
- JTAG chain
- Computer with Windows operation system

### 1.3.2 Software Tools

The following software tools are referenced during this thesis:

- Xilinx ISE Design Suite 14.3
- Xilinx Embedded Development Kit (EDK) 14.3
- Microsoft Visual Studio 2010

## 1.4 Main Steps

In order to reach the previously stated targets, the following steps are scheduled and carried out during the thesis work.

Firstly, an investigation is made to find out the principles and work flows of Xilinx SDK debugging session, which helps to understand where the changes should be made and serves as the foundations of next step. Secondly, a script is written to substitute the original underlying debugging engine of SDK (i.e. GDB), so that when user is debugging via SDK’s graphical debugger interface, the script is able to capture the sequence of commands that SDK sends to GDB, without any interruption or interference of user’s debugging process. Thirdly, a modification should be applied to this script, so that it’s adapted to dual-processor debugging scenarios. Lastly, a few additional functionalities are augmented, so that via the script, the results from both processors can be collected, and the significant digits are calculated and presented to users.

# 2 Recall of Discrete Stochastic Arithmetic (DSA)

In this chapter, the concepts and principles of *Discrete Stochastic Arithmetic* (DSA) are reviewed. In section 2.1, a brief introduction of floating point number representation standard is presented as the first step, and then the round-off error is introduced in section 2.2, afterwards in section 2.3 a brief recall of DSA is shown.

## 2.1 Floating Point Number Representation

Every real number $x$ can be represented as

$$x = s * m * b^{e} \ ,$$

where

- $s$ is the sign bit
- $b$ is the base
- $e$ is the exponent
- $m$ is the mantissa , $1 \leq m < b$, with the form

$m = (d_1 . d_2 d_3 \dots d_n) \ \ \forall i \ \in [1, n], \ \ d_i \in \mathbb{N} \ and \ 0 \leq \ d_i < b$

According to *IEEE Standard for Floating-Point Arithmetic* (IEEE 754), in computer where a floating point number is stored, $b$ is chosen as 2, and therefore it's a sequence of bits made up from 0 and 1, which can be interpreted as:

$$x = s * (d_1 . d_2 d_3 \dots d_n) * 2^{e}$$

where

$$\forall i \ \in [1, n] \ \ d_i \in \{0,1\}$$

In IEEE 754, two most-frequently used binary floating point formats are specified, *single precision* and *double precision* [8]. For single precision floating point number, $n = 24$. As shown in Figure 2.1, it‘s encoded as 32 bits: with first bit as sign bit (0 for + and 1 for -), followed by 8 bits as exponent, and 23 bits as mantissa, which corresponds to $(d_2 d_3 .. d_{24})$, while $d_1$ is hidden, and $d_1 = 1$ for normalized numbers (which is the most case), and $d_1 = 0$ for denormalized numbers. Due to the possibility that the exponent can be negative, the coded exponent results from an addition of the actual exponent and a bias, which is 127 for single precision.

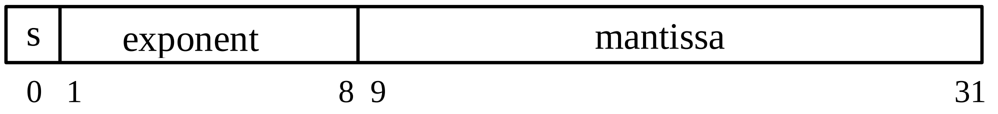


**Figure 2.1**: Single precision floating point number presentation

The double precision floating point number is encoded in a similar way, except $n = 53$, and exponent is encoded as 11 bits, while the bias for exponent is 1023, as depicted in Figure 2.2.

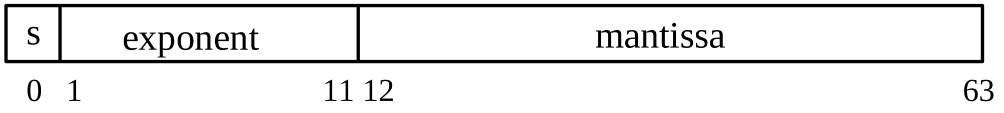


**Figure 2.2**: Double precision floating point number presentation

## 2.2 Rounding Mode

As only finite bits are used to store the floating point numbers, for those real numbers which exceed the maximum length of bits for storage, a rounding operation is necessary.

Let $X$ be a real number in exact arithmetic, then $X$ is bounded by two consecutive floating point numbers, one rounded down $X^-$ and the other rounded up $X^+$, each of them representing the exact representative result [1], i.e. $X^- \leq X \leq X^+$ . Thus, $X$ can be rounded to $X^-$ or $X^+$ depending on which rounding mode is applied.

IEEE 754 defines four such rounding modes, which are:

- **Round to nearest (roundTiesToEven)**: $X$ is rounded to the nearer of $X^+$ or $X^-$. In case that neither is nearer, the even alternative is chosen.
- **Round to zero:** $X$ is rounded to the representable number closer to 0, i.e. $\min\{|X^-|, |X^+|\}$
- **Round to positive-infinity**: $X$ is rounded to $X^+$.
- **Round to negative-infinity**: $X$ is rounded to $X^-$.

*Random rounding*, is when an inexact representable number is obtained and a rounding operation is need, the process to randomly choose $X^+$or $X^-$ with identical probability.

## 2.3 Discrete Stochastic Arithmetic (DSA)

*Discrete Stochastic Arithmetic* (DSA) provides a method for analyzing and controlling round-off errors during the execution of scientific codes. It's an extension of the CESTAC method but also presents new concepts like informational zero, stochastic relations etc, which will be explained afterwards.

The aim of DSA is [4]:

- Detect numerical instabilities
- Evaluate round-off error propagation on each result
- Calculate the accuracy of results in terms of significant bits
- Judge the result is reliable or not

### 2.3.1 CESTAC Method

*Contrôle et Estimation Stochastique des Arrondis de Calculs* (CESTAC) method [12][13] is such a method to evaluate the effect of round-off error propagations and detect numerical instabilities. It was proposed by M. La Porte and J. Vignes in 1974, and the basic principle can be summarized as follows:

1. Run the same piece of codes N times, and *randomly rounding* is applied after each floating point operations.
2. After N executions, N results are gathered and compared.
3. Those parts which are common in all N results are considered to be reliable, and the number of bits of this part is known as *significant digits*.

According to this approach [14], after N times running of the codes, each sample $R_i$ can be modeled as:

$$R_i = r + \sum_{k=1}^{n} g_k(d) 2^{-p} \alpha_k + O(2^{-2p}) \,,$$

where

- $R_i$ : the $i$-th sample, $i \in [1, N]$
- $r$ : the exact result
- $g_k(d)$: quantities depending exclusively on the program and data, but independent of $\alpha_k$
- $\alpha_k$: normalized rounding errors, which are modelled by independent random variables identically and uniformly distributed on (-1,+1)
- $p$: wordlength of mantissa

The reliability of this model and the effectiveness of CESTAC method for actual use in scientific codes can only be guaranteed if the following hypotheses are true [4]:

- Hyp1. The elementary round-off errors $\alpha_k$ of the floating point arithmetic operations are random independent, centered and uniformly distributed variables.

- Hyp2. The approximation of the first order in $2^{-2p}$ is legitimate.

If these two hypotheses hold, then $R_i$, $i \in [1, N]$ is proven to be samples of Gaussian distribution, centered on the exact result $r$, therefore it is possible to use Students test to get a confident interval of $\bar{R}$ with the probability of $(1 - \beta)$ [4][17], where $\bar{R}$ is the average value of N samples, which are given as follows:

$$\bar{R} = \frac{1}{N} \sum_{i=1}^{N} R_i \ .$$

And the precision, i.e. number of significant digits, can be evaluated by the following formula [14]:

$$C_{\bar{R}} = log_{10}\left(\frac{\sqrt{N} * |\bar{R}|}{\tau_\beta * \sigma}\right),$$

where

$$\sigma^2 = \frac{1}{N-1} \sum_{i=1}^{N} (R_i - \bar{R})^2,$$

and $\tau_\beta$ is the critical value of the Student distribution for $N - 1$ degrees of freedom and a probability level $1 - \beta$.

Hypothesis 1 is ensured to be satisfied due to the great universality of the theorem of central limit and robustness of Student law [4], while Hypothesis 2 holds if the terms in $2^{-2p}$ is negligible compared to terms in $2^{-p}$, to be more exact, the following two restrictions must be met:

- The operands of any multiplication are both significant.
- The divisor of any division is significant.

Both of the restrictions are inspected in the implementation of the hardware platform, which will be presented later in Chapter 3.

### 2.3.2 Informational Zero

A result from the CESTAC method is said to be *informational zero* if and only if one of the following two conditions holds:

- $R_i = 0, \forall i \in [1, N]$
- $C_{\bar{R}} \leq 0$

Informational zero is denoted as @.0, from this definition, *Discrete Stochastic Relations* (DSR) can be derived as follows [4]:

Assume $X$ and $Y$ are N-samples provided by CESTAC method,

- discrete stochastic equality (denoted by $s =$) is defined as

$X\ s = Y$ if $X - Y = @.0$

- discrete stochastic inequality (denoted by $s >$ and $s \geq$) are defined as

$X\ s > Y$ if $\bar{X} > \bar{Y}$ and $X - Y \neq @.0$

$X\ s \geq Y$ if $\bar{X} \geq \bar{Y}$ or $X - Y = @.0$

# 3 Hardware Platform Support

In this thesis, an FPGA-based hardware architecture which supports DSA, is served as the hardware platform support for the graphical numerical accuracy debugger. In section 3.1, a general overview of this hardware system is presented, and in the following three sections (section 3.2, section 3.3, section 3.4) the descriptions and functionalities of some key components: Discrete Stochastic Floating Point Unit, Synchronization Unit, and Numerical Accuracy Unit are introduced respectively.

## 3.1 Overview of the Hardware System

The hardware system is located on the Xilinx Virtex-5 FXT ML510 FPGA board. It consists of two hardwired PowrePC440 processors, two *Discrete Stochastic Floating Point Units* (DSFPUs), one *Synchronization Unit* (SyncU), one *Numerical Accuracy Unit* (NAU) and some other necessary components like memories, serial ports etc. The NAU consists of a *Significant Digits Estimation Unit* (SDEU) and a *Numerical Instability Detection Unit* (NIDU).

The overview of this hardware system is shown in Figure 3.1.

Here the PowerPC440 processor, the DSFPU as well as the corresponding memories and other components are said to form a *processing block*. While the synchronization unit, together with NAU, are shared by both processing blocks.

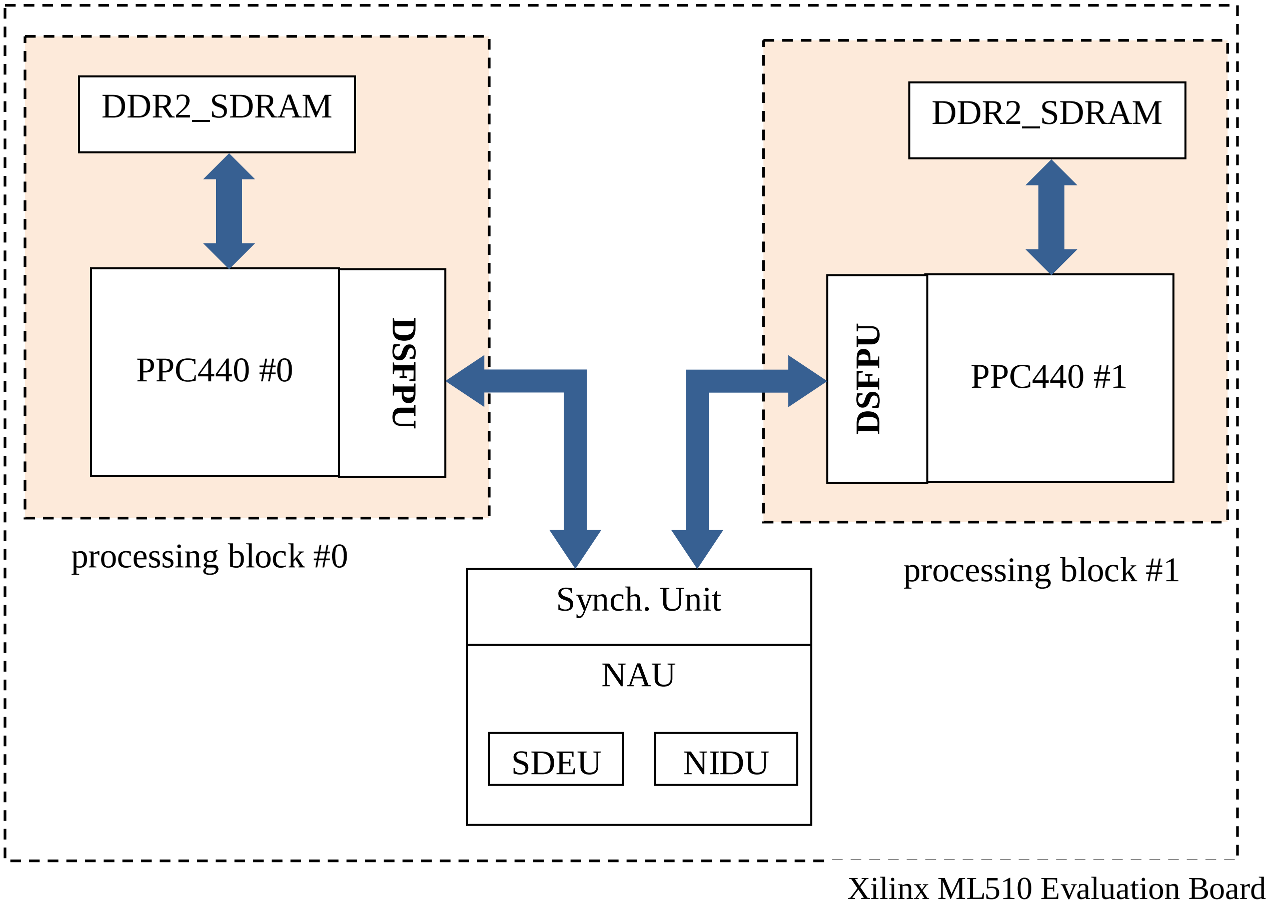


**Figure 3.1**: Overview of the hardware system which support DSA

## 3.2 Discrete Stochastic Floating Point Unit (DSFPU)

The DSFPU, which is connected to PowerPC processors through *Auxiliary Processor Unit* (APU) [7], worked as a coprocessor. Apart from normal functionalities which are in common with traditional IEEE-754 compatible

FPUs (e.g. decoding and execution of the standard floating point operations, support of single precision and double precision formats etc.), it is supposed to support DSA and therefore some more features are added:

- **Random rounding**

  As mentioned in 2.2, random rounding is used in DSA after each floating point operation, to round the result either upwards or downwards randomly with the same probability. In DSFPU, it is implemented by using a *Linear Feedback Shift Register* (LFSR) to generate a pseudo random number [15].

- **Discrete Stochastic Relations support**

  It's implemented by a particular unit to support the DSR which are defined in section 2.3.2. This unit is designed as a common unit for both DSFPUs, because the execution of the program in different processing blocks might jump into different branches of the program depending on their own results obtained. If this is the case, then the subsequent numerical analysis is impossible. Thus, a decision has to be made before the program enters the branch and forwarded both processors, and this discrete stochastic relations unit is designed to generate such a decision.

- **Forward exceptions raised from NAU**

  When there's an exception raised from NAU due to the detection of any numerical instabilities, DSFPU should be able to assert and deassert applicable signals in order to communicate the exception to PowerPC process via APUs properly [7].

## 3.3 Synchronization Unit

According to the principles of DSA, the floating point operations running in each process block need to be synchronously processed. Otherwise, different results of the same variable cannot be collected and subsequent numerical analysis cannot be performed.

Thus the synchronization unit for both processor blocks is necessary. It’s designed in such a way that, when an asynchronous execution is discovered, that is, when a floating point operation on one DSFPU has already started but not on the other, a *stall* signal is issued by this synchronization unit. When such a stall signal is asserted, the DSFPU suspends the current execution by executing stall cycles, keeps all the state unchanged, until the other DSFPU catches it. After that the stall signal is released and both DSFPUs can continue executing.

## 3.4 Numerical Analysis Unit (NAU)

Numerical Analysis Unit (NAU) is the key component of the hardware system with DSA support. It consists of Significant Digits Estimation Unit (SDEU) and Numerical Instability Detection Unit (NIDU), and should have a functional implementation of the following:

a) Estimate the number of significant digits

b) Check the significance of multiplication operands and the divisor, as mentioned in section 2.3.1

c) Check if the accuracy of the result is acceptable, i.e. if the accuracy is lower than the pre-defined threshold

d) Check if there’s a loss of accuracy due to cancellation in addition/subtraction

e) Check if there’s unstable branch

f) Raise the excpetions to DSFPU in case of any detection of numerical instabilities

Function a) is implemented by SDEU, while function b) – f) is implemented by NIDU.

The SDEU is connected to both DSFPUs and calculates the number of significant digits for multiplication operands, divisor, as well as the computed results. An optimized data path for estimation of the exact significant digits for N = 2 (i.e. two processing blocks) is proposed in [15], via this optimization, the cost of hardware resources are also reduced.

After the calculation is done, the computed number of significant digits are sent to Discrete Stochastic Relations Unit (DSRU) to make comparisons for the decision of DSR operations, and/or to the NIDU for the detections of numerical instabilities.

Although the number of significant bits can be calculated in NAU, an extra calculation in the software debugger is required, because:

- The calculation result in NAU is sent to DSRU and/or NIDU for the decision of DSR operations, or for the detection off numerical instabilities. It’s for internal usage and therefore the user cannot obtain this calculation result via debugging interface.
- In order to reduce the cost of hardware resources, the calculations of number of significant bits in NAU is an approximate value.
- It’s not a high demand in terms of calculation speed as it‘s for debugging purpose, therefore calculating in software is sufficient and acceptable.

# 4 SDK Debugging Session

Since the numerical accuracy debugger is based on the FPGA computing system, the FGPA-related Xilinx tools (e.g. XPS, SDK, etc) are referenced. Among them, SDK itself already provides a friendly and convenient graphical debugging interface, with GDB used as the underlying debugging engine. Thus the graphical debugger tool integrated in SDK is here chosen as the starting point of developing the numerical accuracy debugger.

In section 4.1, a few referenced terminologies are explained and the functionalities of used Xilinx tools are introduced, and in section 4.2, the principles and work flows of SDK debugging session are discussed, including single-processor debugging and dual-processor debugging. Based on the analysis on these, a new semi-auto dual-processor debugging flow is proposed and explained in section 4.3.

## 4.1 Xilinx EDK Concepts and Tools

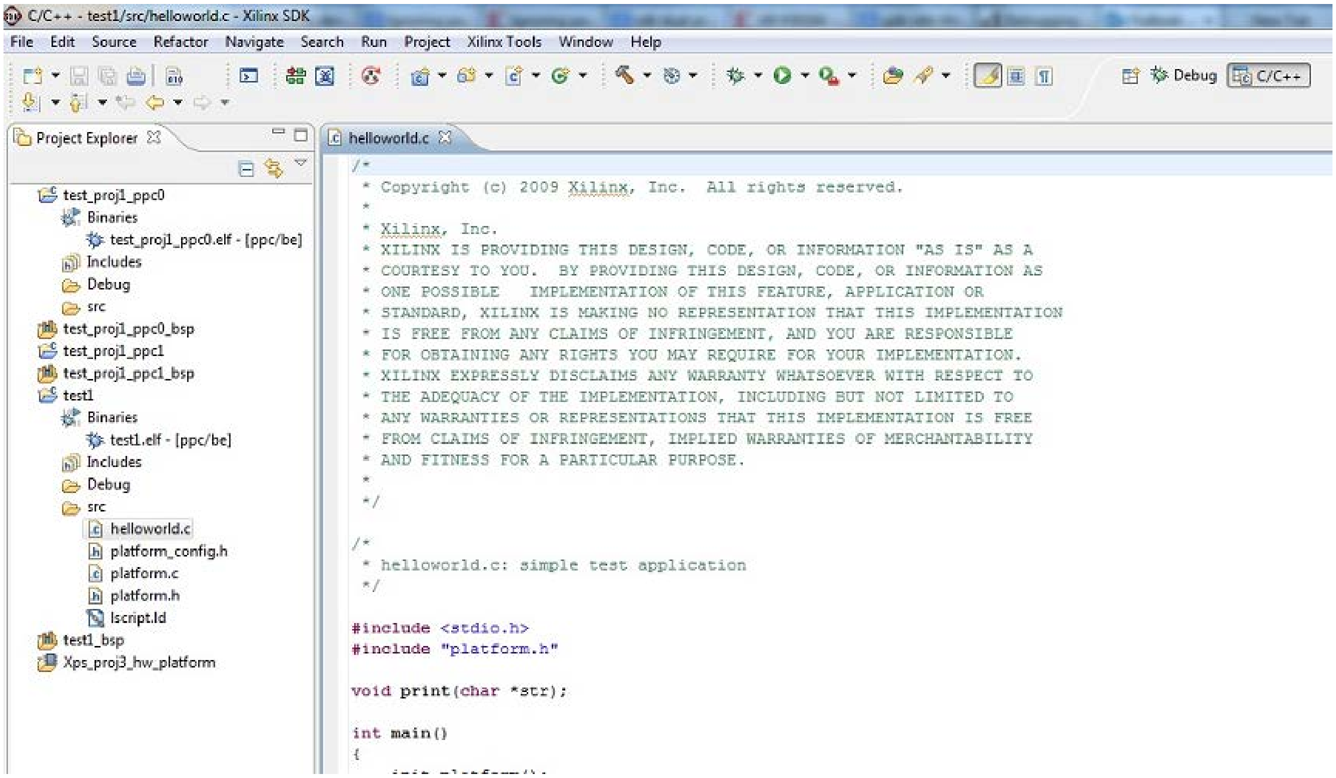
*Xilinx Embedded Development Kit* (EDK) is a collection of tool package, including *Xilinx Platform Studio* (XPS), *Software Development Kit* (SDK), hardware IP and some other components [9]. These tools are designed for the implementation of the complete embedded systems on a Xilinx FPGA device.

### 4.1.1 Software Development Kit (SDK)

While XPS is used for designing and developing the hardware environment of the customized embedded system, and afterwards, this hardware design can be exported to SDK, where the C/C++ embedded software applications running on processors are created and implemented, based on the hardware platform specifications.

Software project is processor-specific, i.e., if more than one processor is specified in the hardware platform implementation, then whenever a software project is created, it must be clearly defined that on which processor would this software project run.

Figure 4.1 shows the screenshot of SDK working environment.



```
/*
 * Copyright (c) 2009 Xilinx, Inc.  All rights reserved.
 *
 * Xilinx, Inc.
 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A
 * COURTESY TO YOU.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS
 * ONE POSSIBLE   IMPLEMENTATION OF THIS FEATURE, APPLICATION OR
 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION
 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE
 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION.
 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

/*
 * helloworld.c: simple test application
 */

#include <stdio.h>
#include "platform.h"

void print(char *str);

int main()
{
```

**Figure 4.1**: SDK working environment

For a software project, the source files, as well as the header files and the *board support package* (BSP) are required, which are listed in the left-side window of the working environment.

The source files, together with necessary header files can be compiled later to result in a binary output (.*elf*) file, which can be downloaded to target processor later for debugging or execution purpose. While the *board support package* (BSP), mandatorily correspond to each software project, is a collection of low-layer drivers and libraries, which are linked by the software application at runtime.

The *C/C++ code perspective* and *debug perspective* are located in the top-right corner. *Perspective* in SDK refers to different displays of windows, and depending on the on-going activities should the perspective change accordingly. When the C/C++ codes are being developped, C/C++ code perspective will be shown, while the binary file is being debugged on hardware, SDK will automatically jump to the debug perspective.

### 4.1.2 Xilinx Microprocessor Debugger (XMD)

Xilinx itself also provides a debugging and verifying tool for the software application running on PowerPC (405 or 440) processor, MicroBlaze processor, or ARM Cortex-A9 MPCore processor [10]. It's so called *Xilinx Microprocessor Debugger* (XMD).

As depicted in Figure 4.2, XMD helps user to debug the software project on hardware by acting as a bridge in between.

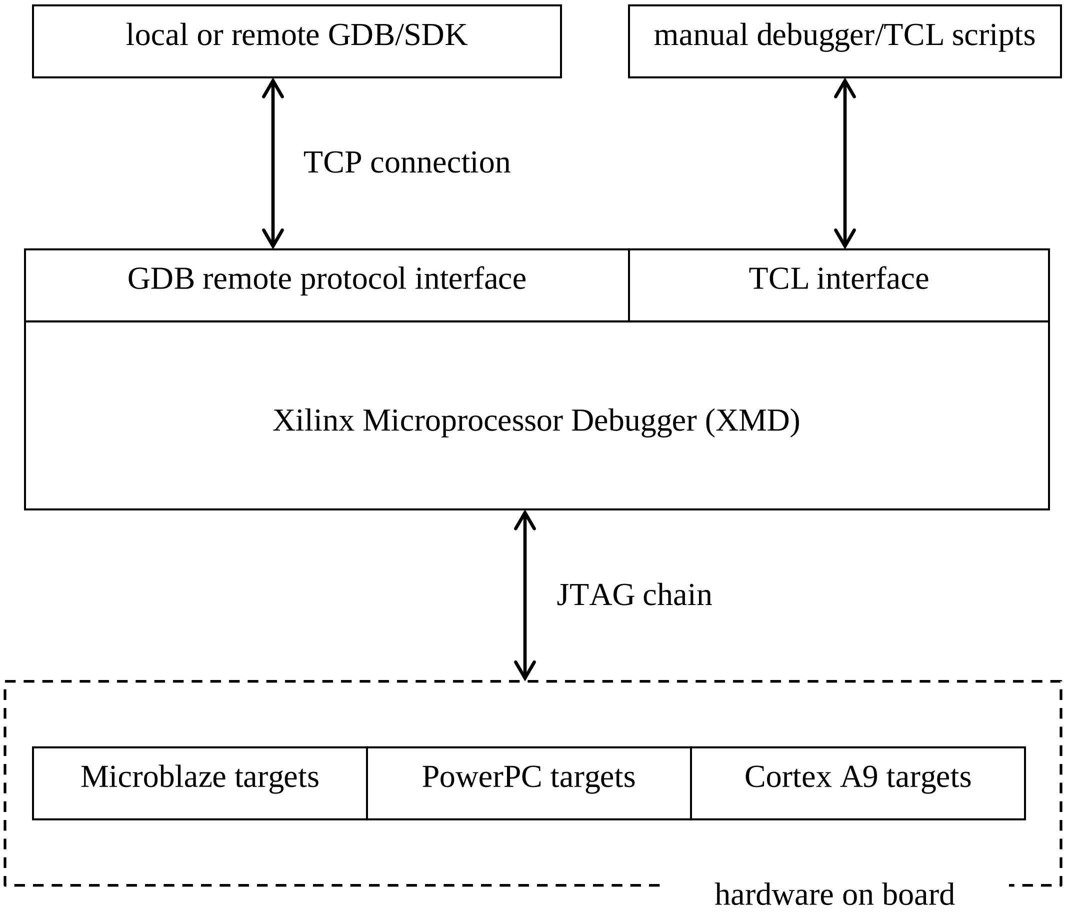


**Figure 4.2**: XMD acts as a bridge

XMD provides a *Tool Command Language* (TCL) interface which can read customized TCL scripts to realize line-control functionalities or commands for debugging, and it also accepts a connection to the local or remote *GNU Debugger* (GDB) via TCP protocol so that the user can control the debugging process on GDB. On the other side, XMD connects to the targets on the actual hardware platform, and allows to download the software applications to hardware targets for debugging or running. These targets can be Microblaze processor targets, PowerPC processor targets, Cortex A9 processor targets, etc.

Beside these, XMD also supports some other interfaces, e.g. socket interface, serial interface, etc., which are not explained in detailed here.

### 4.1.3 GNU Debugger (GDB)

The *GNU Debugger* (GDB), which is one of the most used debuggers, is integrated in SDK and used by SDK as the underlying debugging engine when debugging software applications running on hardware targets.

The GNU Debuggers are classified into different kinds in SDK, depending on which processors they are called for. For debugging the software applications running on Microblaze processor, *mb-gdb* is called; while for those running on PowerPC processor, *powerpc-eabi-gdb* is called.

As mentioned in section 4.1.2, GDB connects to XMD via a remote TCP protocol, and uses XMD as an underlying engine to communicate with the targets on board, which enables remotely debugging from the user's point of view. The detailed work flow of the debugging session will be explained in next section.

## 4.2 Work Flow of SDK Debugging Session

As stated in Chapter 3, the hardware platform is built with PowerPC processors, thus, only debugging session for PowerPC targets is discussed here.

### 4.2.1 Creation of A Test Project

In order to find out the work flow of SDK debugging session, a test project is created as the first step. The hardware system of this test project is shown in Figure 4.3 as a block diagram.

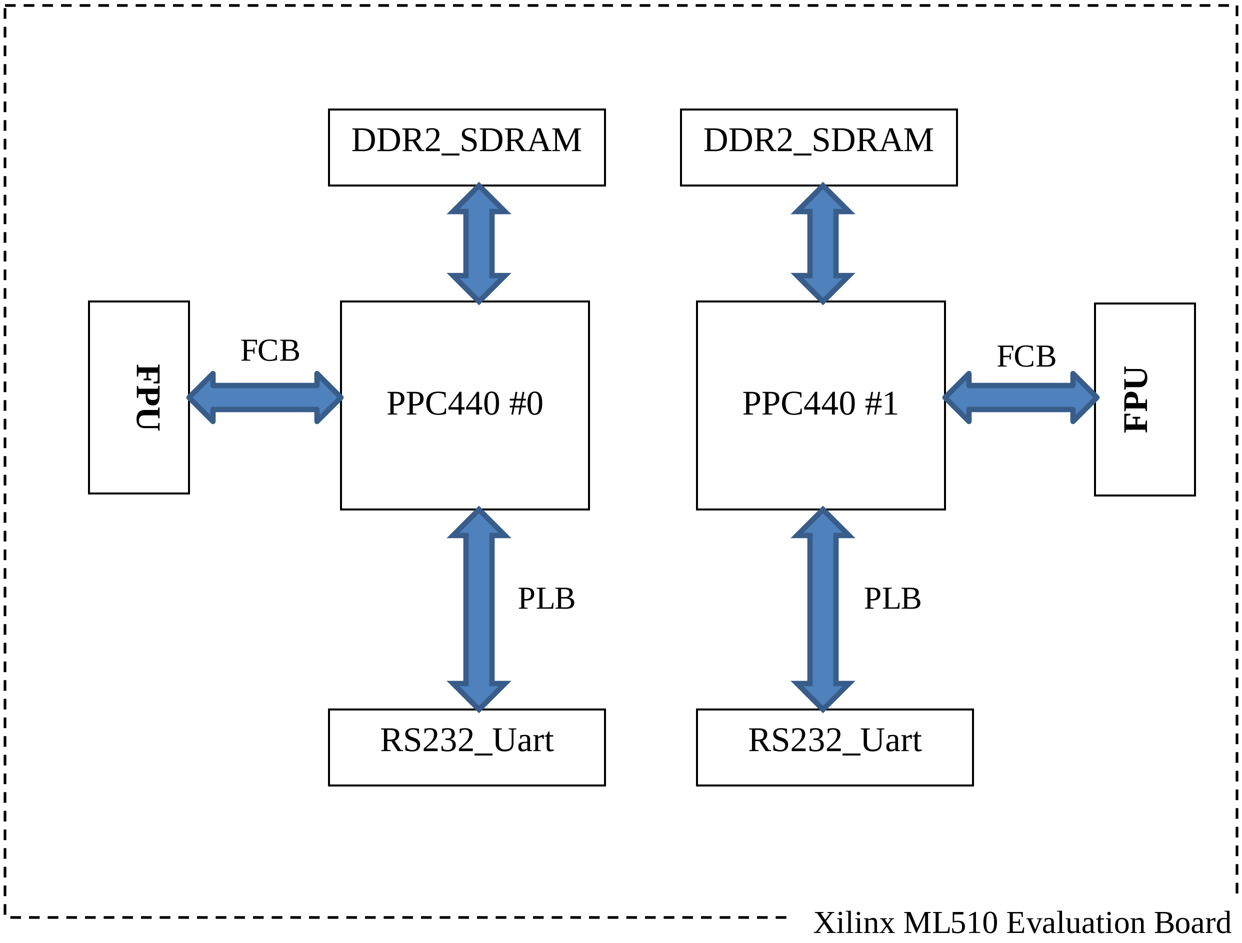


**Figure 4.3**: Hardware system of test project

Actually it's a simplified version of hardware systems presented in Chapter 3: the whole system includes two PowerPC440 processors with maximum operating frequency up to 400MHz, each processor has its own 512MB DDR2_SDRAM attached, and connection to Floating Point Unit (FPU) is established via Fabric Co-processor Bus (FCB), in addition , two RS232_Uart are also connected to PowerPC processors respetively via Processor Local Bus (PLB) so that the output printed results can be observed.

Apart from that, a small piece of C codes are written for the software application, which is shown in Listing 4.1 below.

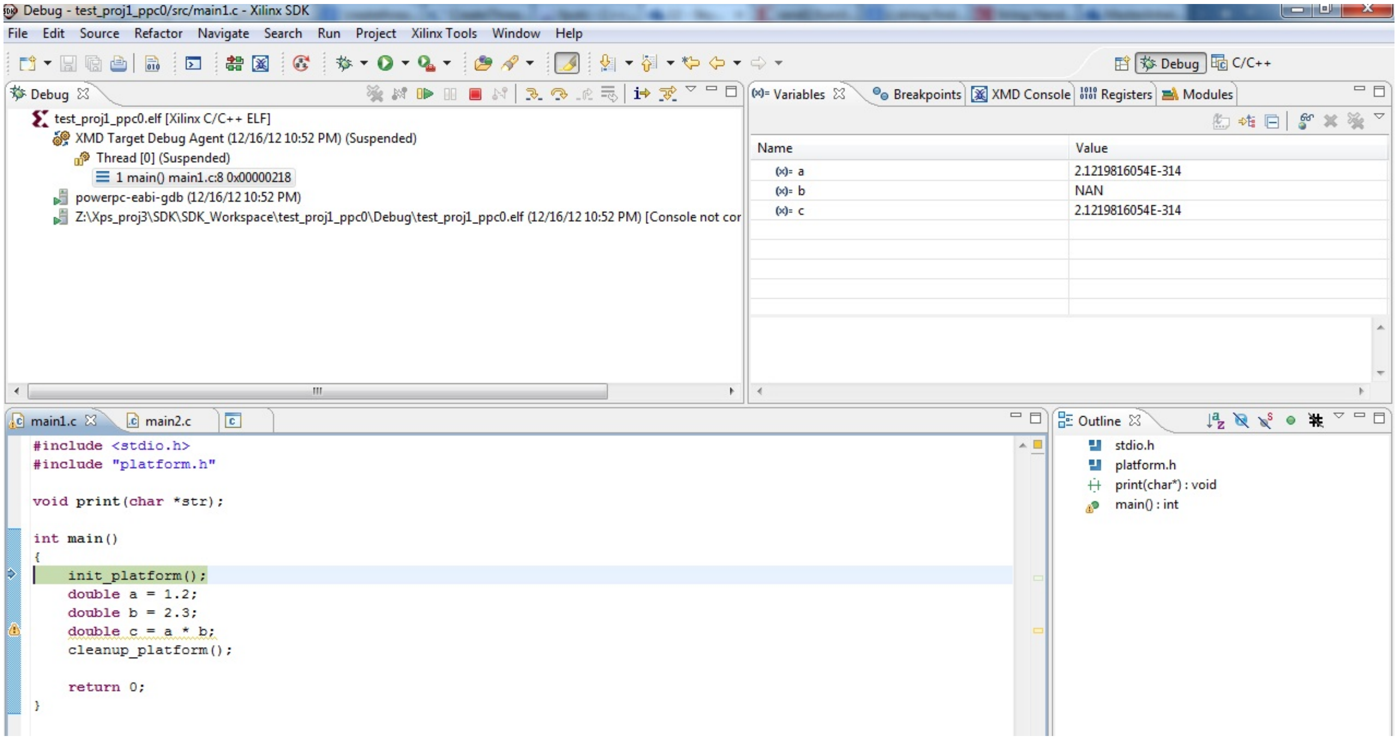
```c
#include <stdio.h>
#include "platform.h"

int main()
{
   init_platform();

   double a = 1.2;
   double b = 2.3;
   double c = a * b;

   cleanup_platform();

   return 0;
}
```

**Listing 4.1**: Software source codes of test project

### 4.2.2 Work Flow of Single-processor Debugging

Let's first consider the *single-processor debugging* scenario. Here single-processor debugging means debug one piece of code on single PowerPC processor, within one debugging session.

Now that the test project is created, the hardware debugging session can be launched (via right click the .elf file and select **Debug as > Launch on hardware**), after several seconds' loading, the debug perspective is presented, and the program is suspended at the beginning of main function, where the first breakpoint is located by default, waiting for the user's next actions. The debug perspective is shown in Figure 4.4.



```
test_proj1_ppc0.elf [Xilinx C/C++ ELF]
  XMD Target Debug Agent (12/16/12 10:52 PM) (Suspended)
    Thread [0] (Suspended)
      1 main() main1.c:8 0x00000218
  powerpc-eabi-gdb (12/16/12 10:52 PM)
  Z:\Xps_proj3\SDK\SDK_Workspace\test_proj1_ppc0\Debug\test_proj1_ppc0.elf (12/16/12 10:52 PM) [Console not cor
```

| Name | Value |
|---|---|
| a | 2.1219816054E-314 |
| b | NAN |
| c | 2.1219816054E-314 |

```
#include <stdio.h>
#include "platform.h"

void print(char *str);

int main()
{
    init_platform();
    double a = 1.2;
    double b = 2.3;
    double c = a * b;
    cleanup_platform();

    return 0;
}
```

Figure 4.4: SDK debug perspective

During the initialization phase of debugging session, two more useful observations are noticed here:

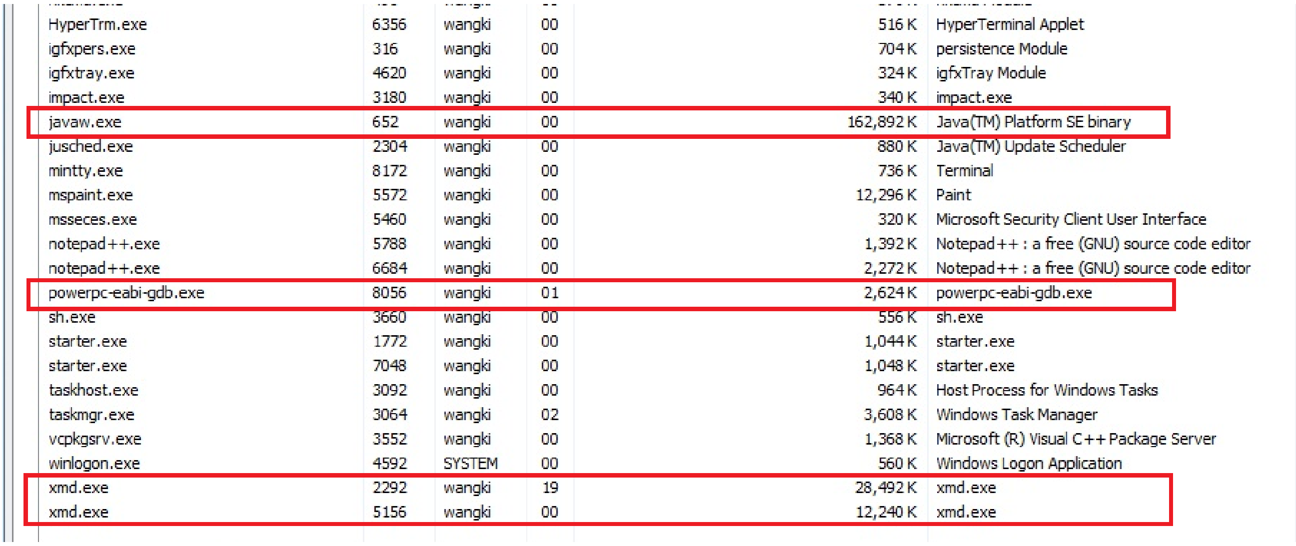
1. Active processes in windows task manager shown in Figure 4.5.

Among them, the remarkable processes are:

- **javaw.exe**
  represents SDK process, as SDK is based on Eclipse.

- **powerpc-eabi-gdb.exe**
  proves that the GDB for PowerPC processor is running, and the actual path can be located via checking the *property* of this process, which turns out to be C:\Xilinx\14.3\ISE_DS\EDK\gnu\powerpc-eabi\nt\bin\powerpc-eabi-gdb.exe

- **xmd.exe**
  there're two xmd.exe listed, whose absolute paths can be both located as:
  C:\Xilinx\14.3\ISE_DS\EDK\bin\nt\xmd.exe
  C:\Xilinx\14.3\ISE_DS\EDK\bin\nt\unwrapped\xmd.exe
  respectively, and it's proven that when SDK is started, the former XMD is called, which will call the latter one afterwards.

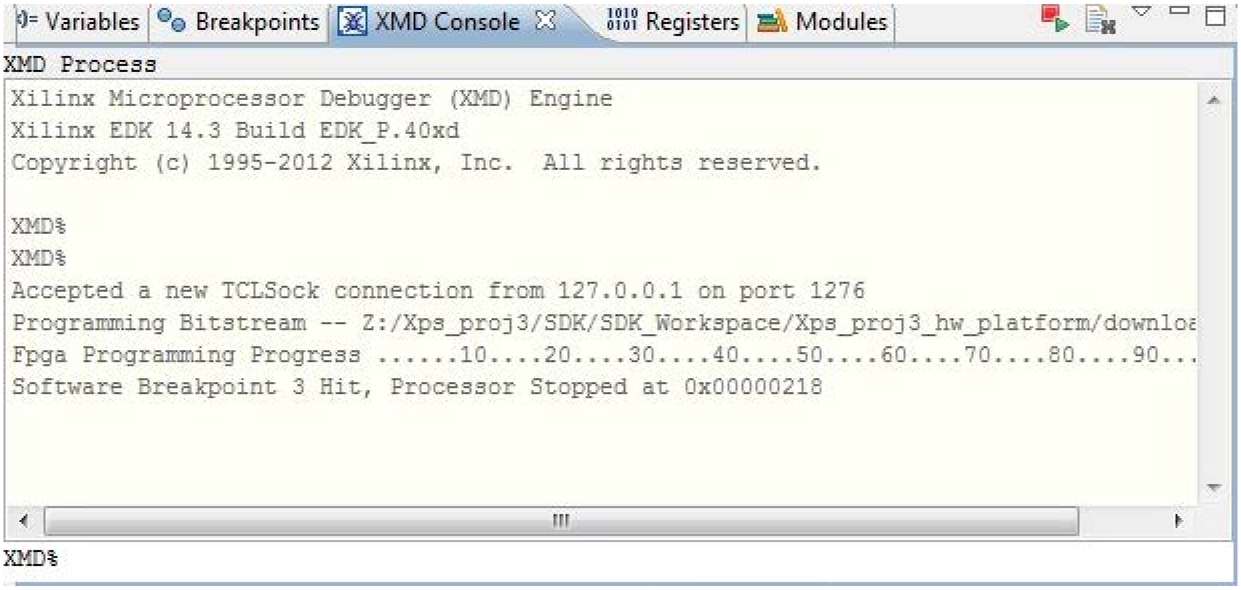


| | | | | | |
|---|---|---|---|---|---|
| HyperTrm.exe | 6356 | wangki | 00 | 516 K | HyperTerminal Applet |
| igfxpers.exe | 316 | wangki | 00 | 704 K | persistence Module |
| igfxtray.exe | 4620 | wangki | 00 | 324 K | igfxTray Module |
| impact.exe | 3180 | wangki | 00 | 340 K | impact.exe |
| javaw.exe | 652 | wangki | 00 | 162,892 K | Java(TM) Platform SE binary |
| jusched.exe | 2304 | wangki | 00 | 880 K | Java(TM) Update Scheduler |
| mintty.exe | 8172 | wangki | 00 | 736 K | Terminal |
| mspaint.exe | 5572 | wangki | 00 | 12,296 K | Paint |
| msseces.exe | 5460 | wangki | 00 | 320 K | Microsoft Security Client User Interface |
| notepad++.exe | 5788 | wangki | 00 | 1,392 K | Notepad++ : a free (GNU) source code editor |
| notepad++.exe | 6684 | wangki | 00 | 2,272 K | Notepad++ : a free (GNU) source code editor |
| powerpc-eabi-gdb.exe | 8056 | wangki | 01 | 2,624 K | powerpc-eabi-gdb.exe |
| sh.exe | 3660 | wangki | 00 | 556 K | sh.exe |
| starter.exe | 1772 | wangki | 00 | 1,044 K | starter.exe |
| starter.exe | 7048 | wangki | 00 | 1,048 K | starter.exe |
| taskhost.exe | 3092 | wangki | 00 | 964 K | Host Process for Windows Tasks |
| taskmgr.exe | 3064 | wangki | 02 | 3,608 K | Windows Task Manager |
| vcpkgsrv.exe | 3552 | wangki | 00 | 1,368 K | Microsoft (R) Visual C++ Package Server |
| winlogon.exe | 4592 | SYSTEM | 00 | 560 K | Windows Logon Application |
| xmd.exe | 2292 | wangki | 19 | 28,492 K | xmd.exe |
| xmd.exe | 5156 | wangki | 00 | 12,240 K | xmd.exe |

**Figure 4.5**: Active processes in windows task manager view

2. Information printed in XMD console window, shown in Figure 4.6.

The message "Accepted a new TCLSock connection from 127.0.0.1 on port 1276" shows that that GDB successfully connects to XMD, while the message "Software Breakpoint 3 Hit, Processor Stopped at 0x000000218" is consistent with the fact shown in Figure 4.4 that the first breakpoint is hit and the processor is temporarily stopped to wait for the user's next operations.



```
Xilinx Microprocessor Debugger (XMD) Engine
Xilinx EDK 14.3 Build EDK_P.40xd
Copyright (c) 1995-2012 Xilinx, Inc.  All rights reserved.

XMD%
XMD%
Accepted a new TCLSock connection from 127.0.0.1 on port 1276
Programming Bitstream -- Z:/Xps_proj3/SDK/SDK_Workspace/Xps_proj3_hw_platform/downloa
Fpga Programming Progress ......10....20....30....40....50....60....70....80....90...
Software Breakpoint 3 Hit, Processor Stopped at 0x00000218
```

```
XMD%
```

**Figure 4.6:** Information printed in XMD console window

According to further reading in [10] and conclusion from the above observations, it's proven that whenever XMD connects to a hardware target on board, it opens a GDB server, together with a listening port (port number in default: *1234*), which allows a local or remote connection from GDB via this TCP port, this SDK single-processor debugging/connection flow is shown in Figure 4.7:

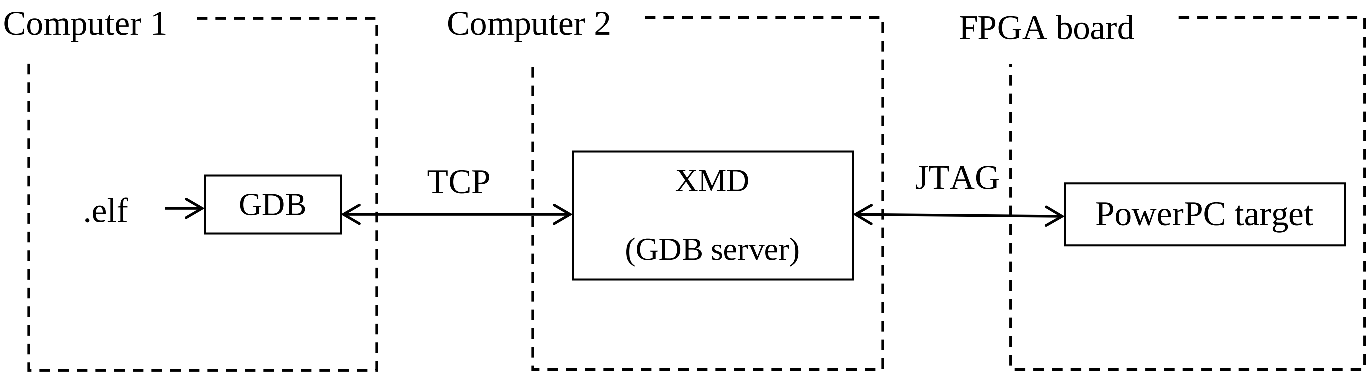


**Figure 4.7:** SDK single-processor debugging/connection flow

Here GDB and XMD are automatically called by SDK once the “debug on hardware” command is received from the user. In the flow graph, GDB and XMD are distributed on two different computers, which actually realize a remote-debugging functionality. However, they can also both located on the same computer (i.e. Computer 1 and 2 in Figure 4.7 are the same computer), which is adopted in our case for convenience purpose.

As a conclusion, when **Debug as > Launch on hardware** is applied to .elf files, a series of operations are handled by SDK in background during the debugging session launch and initialization period, which are listed below:

- **Connect XMD to hardware target with the command**
  ```
  connect ppc hw –debugdevice devicenr x cpunr x
  ```
  This will open a gdb server on XMD and a TCP port for GDB connection.

- **Execute powerpc-eabi-gdb.exe with the commnd**
  ```
  powerpc-eabi-gdb [options] –nw testelf.elf
  ```
  The actual options will be discovered later.

- **Connect GDB to XMD via the command**
  ```
  target remote localhost:1234
  ```
  Here *localhost* means the GDB and XMD are located on the same machine, while *1234* represents the TCP port opened by XMD. After the command is successfully called, XMD will also print out a confirmation message that the GDB connection is accepted.

- **Dowland .elf file to board**
  which is equal to XMD command `dow test.elf`

- **Set breakpoints, initialize debug information**
  breakpoints are set in the beginning and end of main function by default, then the user‘s customized breakpoints are added

- **Enter the debug perspective**

Notice: all of these operations are automatically done by SDK, there’s no manual input or commands from user point of view at all.

### 4.2.3 Work Flow of Dual-processor Debugging

Now let's consider the dual-processor debugging scenario.

In fact, it is possible to connect single XMD instance to both PowerPC targets at the same time, and switch between different targets is also possible, as depicted in Figure 4.8:

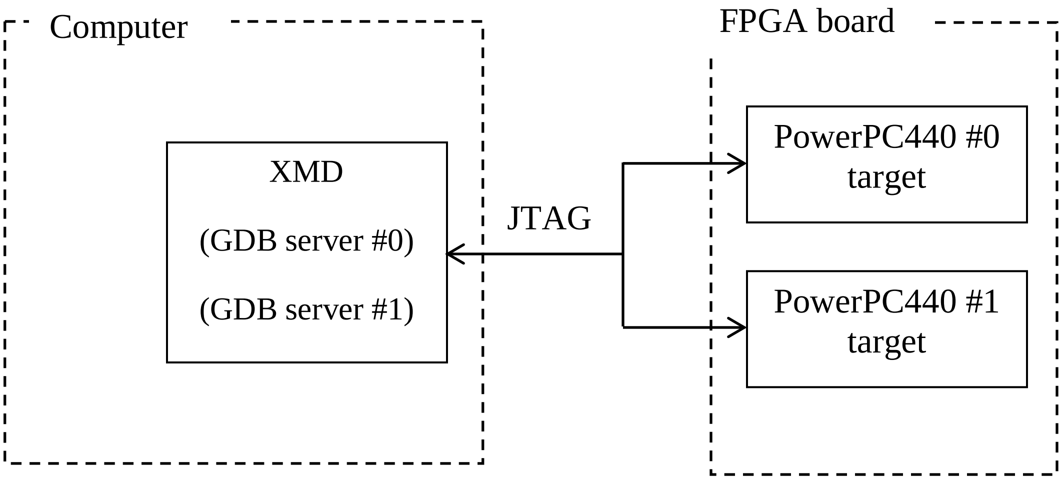
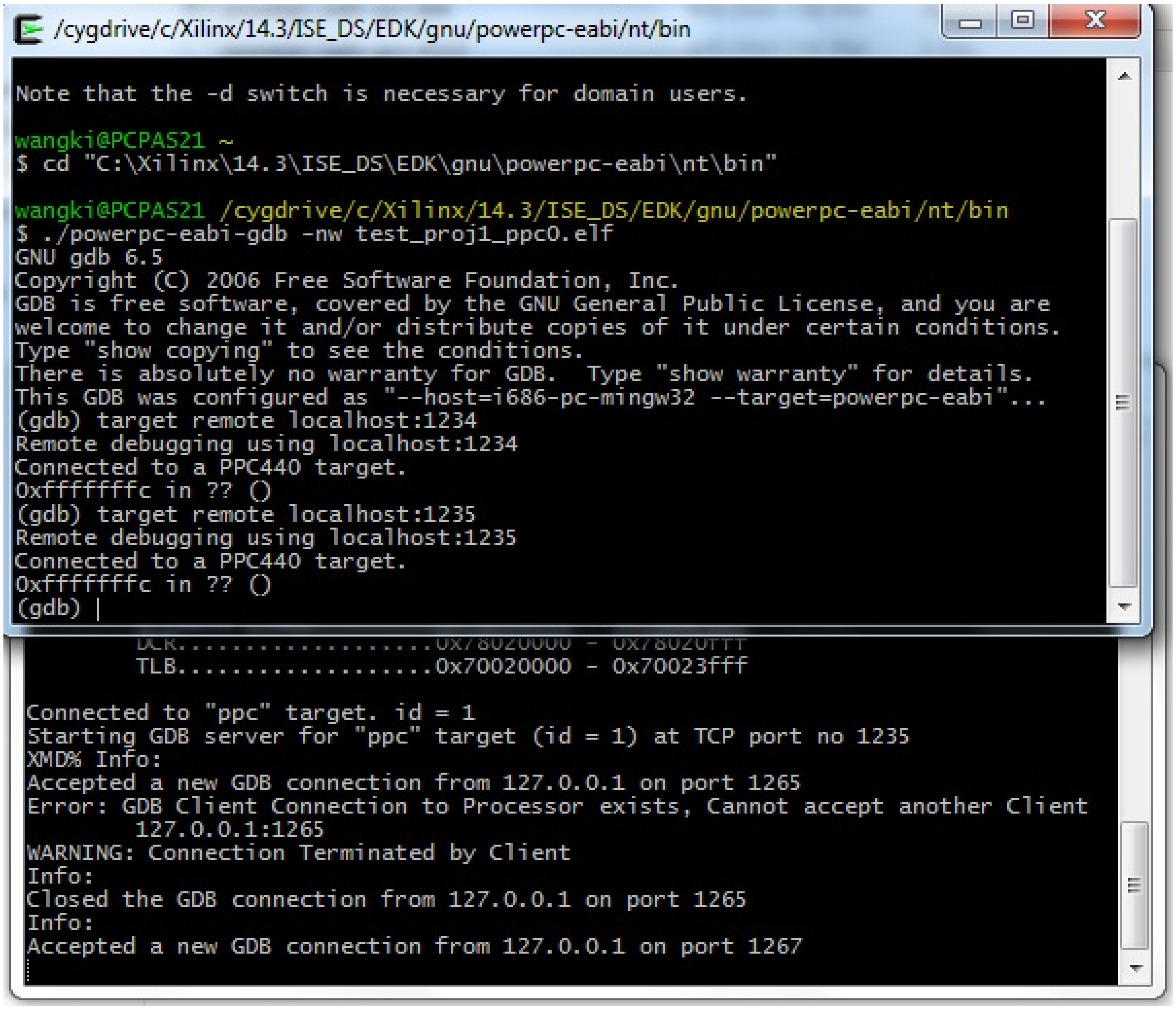


**Figure 4.8:** XMD connects to both PowerPC440 targets

By connecting to both processors, XMD will open two GDB servers and two listening ports, according to the principles explained in section 4.2.2. However, it's impossible for single GDB instance to connect to both GDB servers: when the same GDB instance, which is already connected to one GDB server opened by XMD, is forced to connect to the other GDB server opened by XMD, it will be shown that the previous GDB connection is closed automatically, as depicted in Figure 4.9.

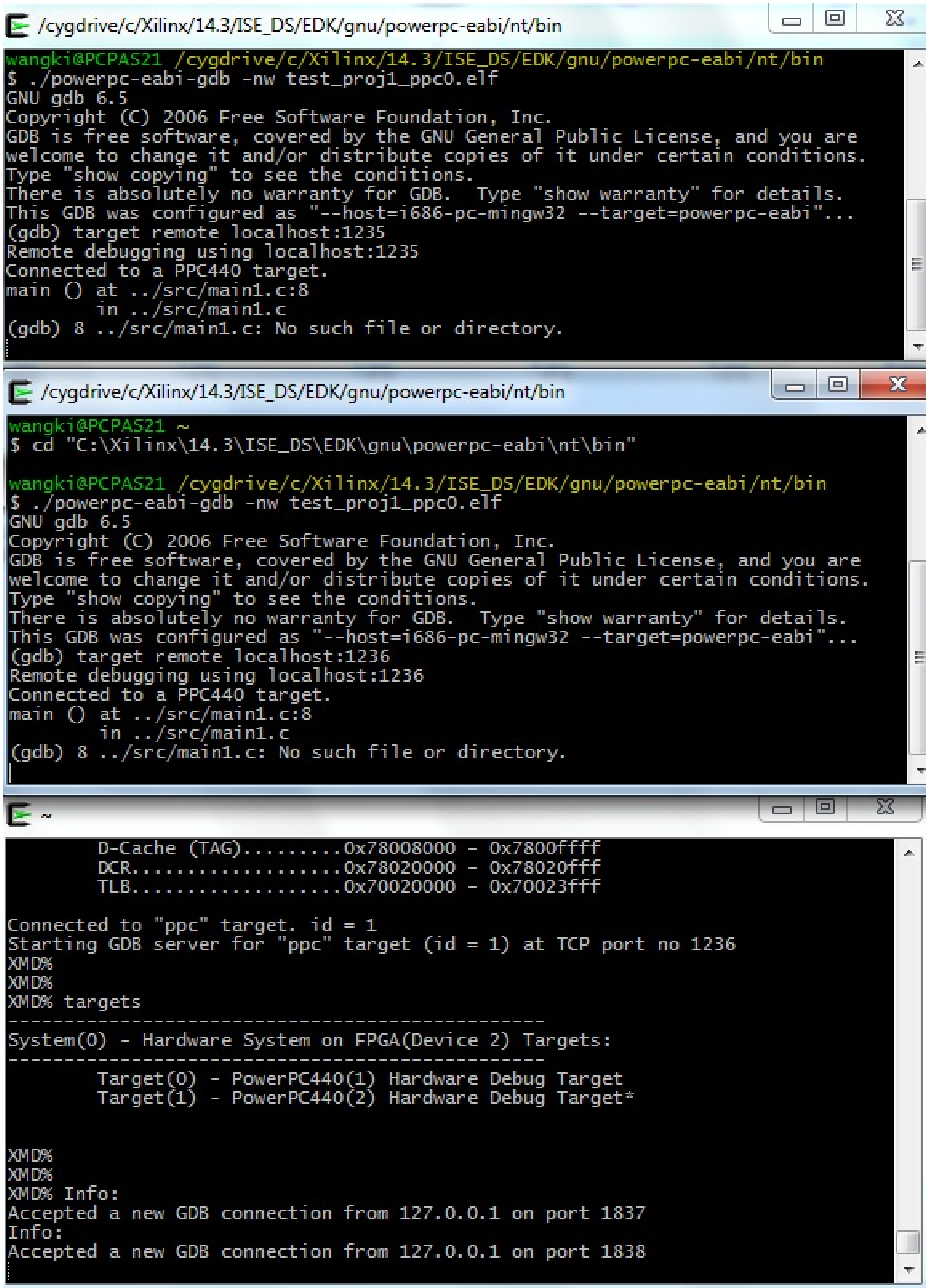


```
Note that the -d switch is necessary for domain users.

wangki@PCPAS21 ~
$ cd "C:\Xilinx\14.3\ISE_DS\EDK\gnu\powerpc-eabi\nt\bin"

wangki@PCPAS21 /cygdrive/c/Xilinx/14.3/ISE_DS/EDK/gnu/powerpc-eabi/nt/bin
$ ./powerpc-eabi-gdb -nw test_proj1_ppc0.elf
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-mingw32 --target=powerpc-eabi"...
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
Connected to a PPC440 target.
0xfffffffc in ?? ()
(gdb) target remote localhost:1235
Remote debugging using localhost:1235
Connected to a PPC440 target.
0xfffffffc in ?? ()
(gdb) |
```

```
        TLB..................0x70020000 - 0x70023fff

Connected to "ppc" target. id = 1
Starting GDB server for "ppc" target (id = 1) at TCP port no 1235
XMD% Info:
Accepted a new GDB connection from 127.0.0.1 on port 1265
Error: GDB Client Connection to Processor exists, Cannot accept another Client
        127.0.0.1:1265
WARNING: Connection Terminated by Client
Info:
Closed the GDB connection from 127.0.0.1 on port 1265
Info:
Accepted a new GDB connection from 127.0.0.1 on port 1267
```

**Figure 4.9:** XMD closes one GDB connection

However, this doesn’t mean that XMD cannot accept two GDB connections at the same time. It is possible, but only if two GDB instances are used for connection. Figure 4.10 shows the situation.



```
wangki@PCPAS21 /cygdrive/c/Xilinx/14.3/ISE_DS/EDK/gnu/powerpc-eabi/nt/bin
$ ./powerpc-eabi-gdb -nw test_proj1_ppc0.elf
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-mingw32 --target=powerpc-eabi"...
(gdb) target remote localhost:1235
Remote debugging using localhost:1235
Connected to a PPC440 target.
main () at ../src/main1.c:8
        in ../src/main1.c
(gdb) 8 ../src/main1.c: No such file or directory.
```

```
wangki@PCPAS21 ~
$ cd "C:\Xilinx\14.3\ISE_DS\EDK\gnu\powerpc-eabi\nt\bin"

wangki@PCPAS21 /cygdrive/c/Xilinx/14.3/ISE_DS/EDK/gnu/powerpc-eabi/nt/bin
$ ./powerpc-eabi-gdb -nw test_proj1_ppc0.elf
GNU gdb 6.5
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-mingw32 --target=powerpc-eabi"...
(gdb) target remote localhost:1236
Remote debugging using localhost:1236
Connected to a PPC440 target.
main () at ../src/main1.c:8
        in ../src/main1.c
(gdb) 8 ../src/main1.c: No such file or directory.
```

```
        D-Cache (TAG).........0x78008000 - 0x7800ffff
        DCR...................0x78020000 - 0x78020fff
        TLB...................0x70020000 - 0x70023fff

Connected to "ppc" target. id = 1
Starting GDB server for "ppc" target (id = 1) at TCP port no 1236
XMD%
XMD%
XMD% targets
------------------------------------------------
System(0) - Hardware System on FPGA(Device 2) Targets:
------------------------------------------------
        Target(0) - PowerPC440(1) Hardware Debug Target
        Target(1) - PowerPC440(2) Hardware Debug Target*


XMD%
XMD%
XMD% Info:
Accepted a new GDB connection from 127.0.0.1 on port 1837
Info:
Accepted a new GDB connection from 127.0.0.1 on port 1838
```

**Figure 4.10:** XMD successfully accepted two GDB connections

Based on this, if two software projects are created for two PowerPC processors respectively, SDK allows the two .elf files to be debugged on different PowerPC processors simultaneously but independently, which is shown in Figure 4.11:

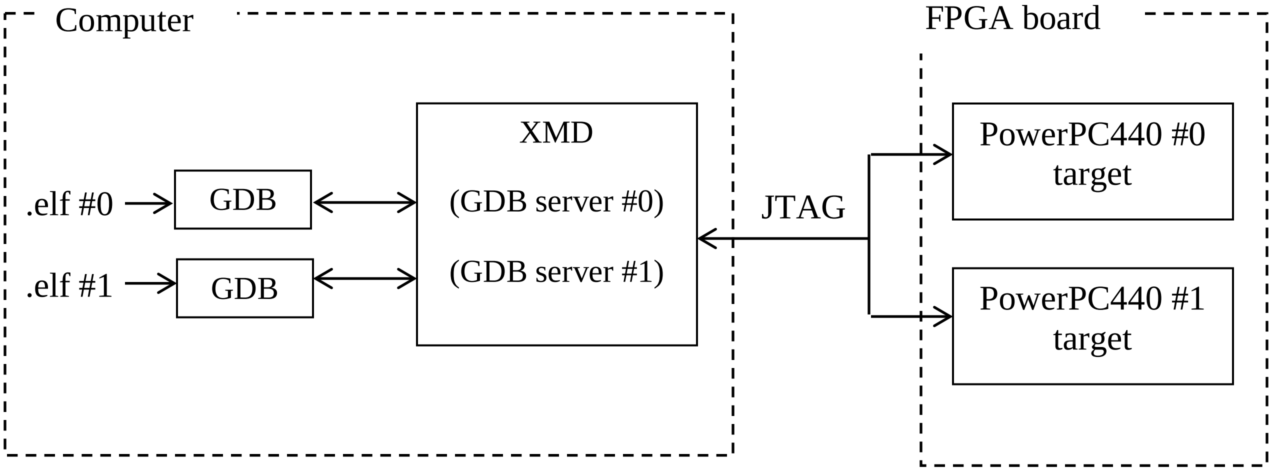


**Figure 4.11:** SDK dual-processors debugging/connection flow

Here “simultaneously but independently” means that the two debugging sessions can be proceeded in parallel, but operations/commands which the user performs in .elf #0 debugging session will not affect the user’s operations/commands in .elf #1 debugging session, they are *asynchronously* proceeded. The debugging status (breakpoints, variable values) will not take effect in each other either. Moreover, users can switch back and forth between these two debugging sessions freely.

However, the pre-set debugging modes mentioned above are not the debugging flows required, because firstly, the same piece of the code needs to be debugged; secondly, the debugging operations have to be synchronously performed, which means, for instance, when user asks PowerPC #0 to do “*step over*” operations, the same commands should be received and carried out by PowerPC #1, only in this case can the results be obtained from both processors after each floating point operation.

One possible solution is creating two software projects for both PowerPCs with exactly the same source codes, and the same debugging operations are repeated manually in both debugging sessions. However, it would be obviously too much work, when the source codes grow in larger-size and the debug operations also increase. The user has to set exactly the same breakpoints, print out exactly the same variable values, step into and over exactly the same functions/statements, which is not only troublesome, but also easy to make mistakes.

In order to overcome this problem, a new and feasible dual-processor debugging flow must be developed.

## 4.3 A Semi-auto Dual-processor Debugging Flow

As discussed in section 4.2.3, the automatic SDK dual-processor debugging flow will not satisfy our target. Thus in this section, a combination of SDK debugging and manual connection is proposed as the new dual-process debugging flow here.

It is based on the fact that in SDK debugging session, SDK will translate each user interface action in debug perspective (for instance, press the button "*step over*", double click to set breakpoints, etc.) into a sequence of GDB commands, while process the output of GDB to display the current state of the program (e.g., values of variables, listing of breakpoints/registers) being debugged [18]. According to the principles, a semi-auto dual-processor debugging flow is depicted in Figure 4.12 as follows:

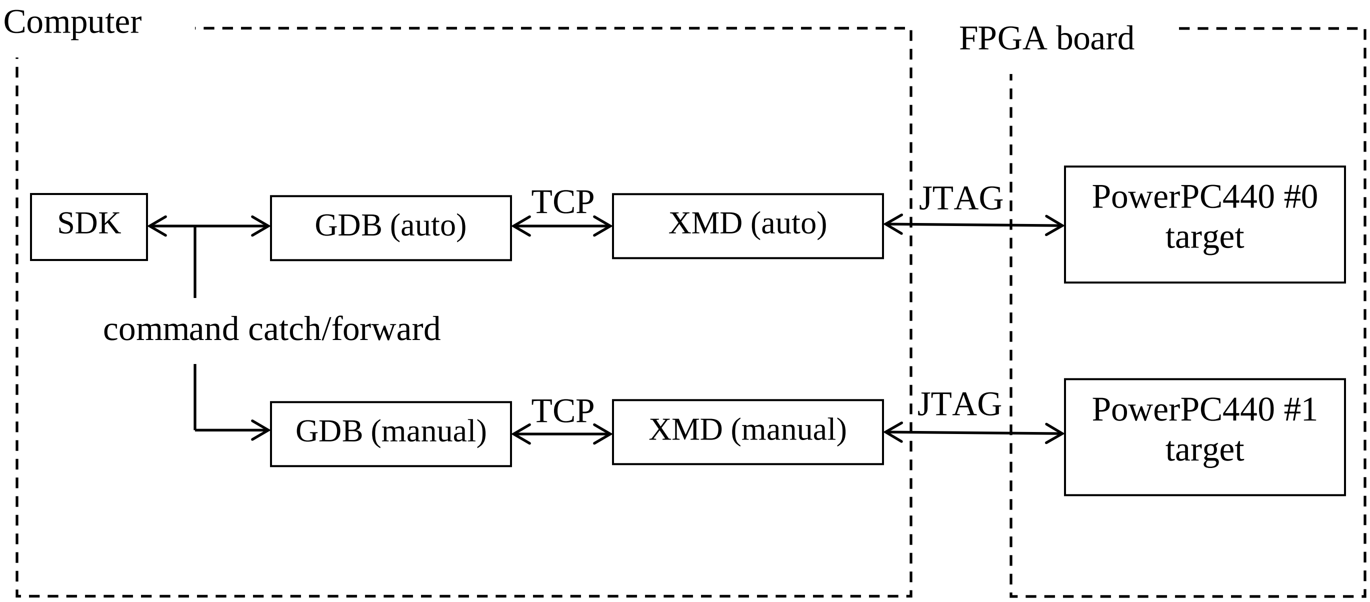


**Figure 4.12:** Semi-auto dual-processor debugging flow

As shown in Figure 4.12, the semi-auto debugging flow consists of two paths, the upper path is handled by SDK, after the “*Launch debugging session on hardware*” command is received by SDK, it will automatically call XMD to connect to PowerPC target, download .elf file, and execute powerpc-eabi-gdb.exe to connect to XMD and so on, as all described in section 4.2.2. While in the lower path, the user has to set up the connections himself before the debugging session is actually started.

Therefore, from SDK point of view, it looks like “single-processor debugging”, as exactly one .elf file is asked to debug on only one PowerPC processor. However, in fact, the “dual-processor debugging” functionality is realized by adding the manual path.

Compared to Figure 4.11, there’s one more difference here: another XMD instance is used, this is based on the fact that we cannot ask the XMD in the auto path to connect to both PowerPCs, because XMD (auto) is handled by SDK, and SDK will only connect it to one PowerPC processor if SDK considers it as a “single-processor debugging” case.

The key point here is the “*command catch/forward*” functionality. The goal of this part is, when user is performing debug operations in SDK and SDK converts the user’s actions into a series of GDB commands, these commands are captured in the half-way, copied, and sent to GDB in the manual connection path. Through this method, the user only has to do debug operations once, but results in the emission and reception of GDB commands in both GDBs, followed by the transmission to XMD and applied to both PPC440 targets. This will meet our target in the right way: debug the same program synchronously in both processors.

Thus, the implementation of this “*command catch/forward*” functionality would be the next point of discussion, which is presented in next chapter.

# 5 Implementation of the Semi-auto Dual-processor Debugging Flow

As discussed in section 4.3, in order to realize the proposed semi-auto dual-processor debugging flow, the implementation of command catching functionality must be investigated as the first step; this is done by a substitution to the original GDB. In section 5.1, a basic concept and principle for this implementation is explained, while in section 5.2, the GDB input commands catching functionality is explained in details. After that, the implementation is extended to dual-processor debugging scenario in section 5.3. In section 5.4, the GDB output message catching functionality is added, and in section 5.5, the way of gathering the results and computing the significant bits is presented.

## 5.1 Basic Principle for Implementation

As mentioned in section 4.2.3, SDK will convert the user's actions in a sequence of GDB commands which are sent to powerpc-eabi-gdb.exe, and process the output of GDB to update the display of current state of the program in the graphical SDK debug perspective. Therefore, the most important issue is to capture the sequence of GDB commands which are sent from SDK to GDB.

In order to capture the commands, we should either focus on the output stream of SDK, or the input stream of GDB (simply because the commands are transmitted from SDK to GDB). Considering SDK will not only send commands to GDB, but also probably to XMD or other components, and also sdk.exe might be well coded to interact with other executable programs, which makes it difficult to be modified or substituted. Therefore, the input stream of GDB should be the target for investigation.

The basic idea is to write a new powerpc-eabi-gdb.exe, to substitute the original one. Whereas in the newly written GDB, it catches the received commands, performs necessary processing, and forwards them to the original one so that the original GDB can deal with all the tasks which it is supposed to do. By this means the new GDB is well pretended, that is, from SDK point of view there're no changes made Figure 5.1 shows this basic idea:

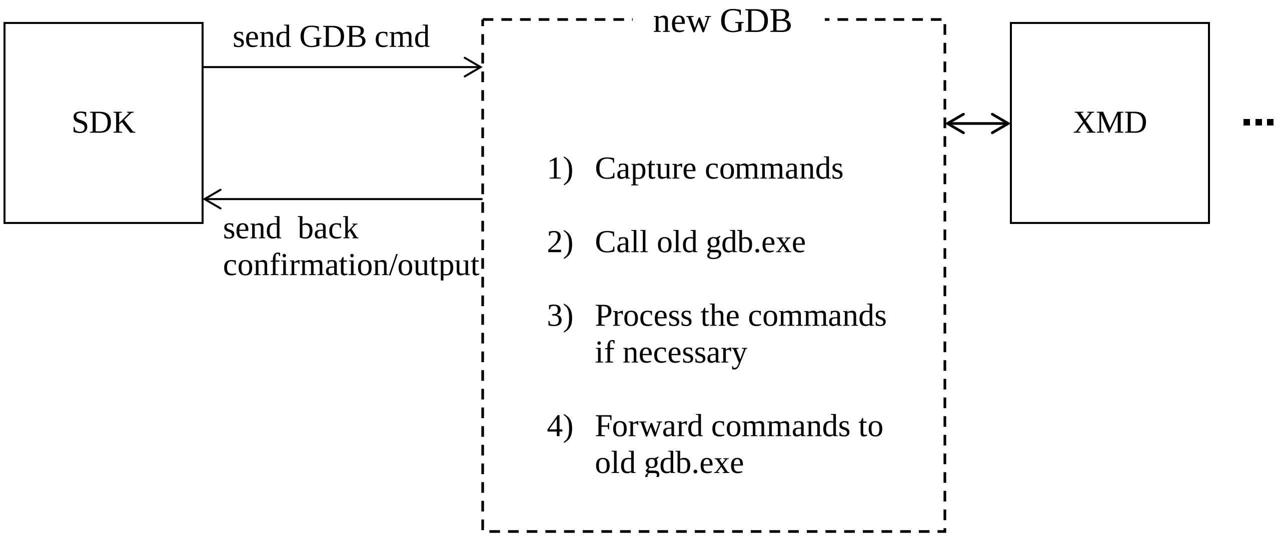


**Figure 5.1:** Basic idea of GDB substitution

Here “send back confirmation/output” means that GDB has to send some confirmation message or output message which will be processed by SDK. However, GDB is still connected to XMD, as depicted Figure 4.13, there’s no conflict between them.

## 5.2 Command Catching/Forwarding

### 5.2.1 Arguments Passing

According to Figure 4.5, the actual GDB called by SDK during the debugging session can be located:

C:\Xilinx\14.3\ISE_DS\EDK\gnu\powerpc-eabi\nt\bin\powerpc-eabi-gdb.exe

This is therefore the right GDB which should be substituted (will be called gdb.exe for short in the following). It can be renamed as powerpc-eabi-gdb-orig.exe (i.e. the original GDB, will be called gdb-orig.exe for short in the following), while the newly written GDB takes the name powerpc-eabi-gdb.exe which can be recognized by SDK. Both GDBs are located under C:\Xilinx\14.3\ISE_DS\EDK\gnu\powerpc-eabi\nt\bin.

Since we are only interested in the commands that GDB receives, the arguments when SDK calls GDB, however, should remain unchanged and passed to the original GDB. This arguments passing functionality is implemented simply by copying the arguments of main() function and passing them when calling old GDB. Furthermore, these arguments can be printed out, which is shown in Listing 5.1, and a glance can be casted over them:

```
powerpc-eabi-gdb
-q
-nw
-i
mi
--cd=Z:\Xps_proj3\SDK\SDK_Workspace\test_proj1_ppc0
--command=.gdbinit
Z:\Xps_proj3\SDK\SDK_Workspace\test_proj1_ppc0\Debug\test_proj1_ppc0.elf
```

**Listing 5.1:** The arguments which SDK passes to GDB

The meanings of the options can be examined in [19]. Among them the remarkable option here is –i mi, where mi stands for *machine interface*. This indicates that the commands that GDB receives from SDK are GDB/MI commands, which is a bit different from normal GDB debugging commands syntax. For more information about GDB/MI interface, please refer to [20].

### 5.2.2 GDB Input Stream Reading Model

In order to read information from the input stream of GDB, the property of *standard input* (STDIN) of GDB, when called by SDK, must be invested. Here *property* stands for the type of input stream of GDB, it might be the input from keyboard, a console input buffer, a reading end of a pipe, etc.

Obviously, the STDIN of GDB cannot be the keyboard input: the user doesn't need to input any characters from keyboard during the debugging session. Therefore the possibility of a console input buffer is discussed in the following.

A *console* is an interface that provides I/O to character-mode application, and a console consists of one *input buffer* and one or more *screen buffers* (output buffer) [21]. A console is created when a *console process* is invoked. A *console process* is a character -mode process whose entry point is the main() function [22], for example, the windows command processor is such a console process, when invoked, a console is created as well. If the user wants to call other console processors from the command processor window, one can specify whether the new process should inherit the parent processor's (command processor) console, or a new console should be created for the new process.

In addition, a process can be attached to at most one console, on the other side, one console can be attached with multiple processes. When a new console is created, the console's input and output buffers are created as well, which serve as the default standard input (STDIN), standard output (STDOUT)/standard error (STDERR) of the attached process, respectively.

Although there's no knowledge about how GDB is called by SDK, it can still be proved via AllocConsole() function that GDB is a console process which has been already attached with a console, an error message, shown in Listing 5.2, will be printed if AllocConsole() function is applied to the source code of new GDB:

```
ERROR: API = Allocate console
error code = 5
message    = Access is denied.
```

**Listing 5.2:** Error message when AllocConsole() is applied

AllocConsole() function only fails when the calling process already has a console attached. This proves that when GDB is called by SDK, it’s already attached with a console, which is hidden in front of users though.

However, this fact doesn't say anything about the STDIN of GDB. It should be the console input buffer in default case, but also can be redirected to somewhere else. In order to invest it in depth, GetConsoleMode() function, together with other functions are applied here, which is shown in Listing 5.3:

```
//check whether the console input buffer is the STDIN of the program
HANDLE hConIn, hStdin;
SECURITY_ATTRIBUTES sa;
DWORD ConMode = 0x0;

hStdin = GetStdHandle(STD_INPUT_HANDLE);
hConIn = CreateFile("CONIN$",GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ,&sa,OPEN_EXISTING,NULL,NULL);

if(!GetConsoleMode(hStdin, &ConMode))
                    DisplayError("Get Console Mode");
```

**Listing 5.3:** Source codes to check the console input buffer

Here hConIn, which is returned by CreateFile() function,  is ensured to be is the handle of console input buffer, even though the STDIN of the program might be redirected to other handles, while hStdin, which is returned by GetStdHandle() function,  is ensured to be the handle of input buffer of the calling process, i.e., the handle after redirection in case there’s I/O redirection involved. If it can be proved that hConIn and hStdin points to the same handle object, then it can be concluded that the STDIN of GDB is just the console input buffer.

There’re a few methods to examine this, here GetConsoleMode() function is employeed. By applying GetConsoleMode(hStdin, &ConMode) , an error message occured , which is listed in Listing 5.4 below:

```
ERROR: API = Get Console Mode
error code = 6
message    = The handle is invalid.
```

**Listing 5.4:** Error message when GetConsoleMode() is applied

Since GetConsoleMode() function only accepts the handle of console input buffer as the first argument, thus, the “The handle is invalid” error message indicates that hStdin is not the handle accepted, i.e., not the handle of console input buffer, which comes into the conclusion that the STDIN of GDB is not the console input buffer, which results in that console I/O functions ( ReadConsole(), WriteConsole(), etc ) cannot be applied.

From the discussions above, it is clear that when SDK invokes GDB, the STDIN of GDB is already redirected by SDK. Figure 5.2 presents this situation:

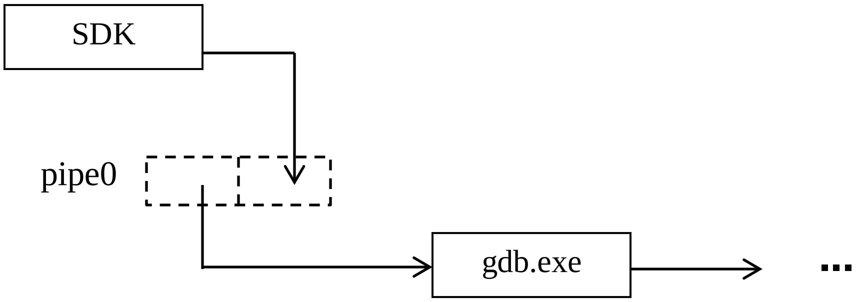


**Figure 5.2:** Redirected-STDIN GDB communicates with SDK

We don't know where the STDIN of GDB is exactly redirected to, however, it is not an important issue, as long as the redirected standard input handle can be obtained by GetStdHandle() function and used for reading data.

Thus here the pipe structure is assumed: SDK sends the commands to the writing-end of pipe0, while GDB reads the commands from the corresponding reading-end of the pipe.

Similarly, a pipe structure is also considered to be applied to the STDOUT/STDERR of GDB. That is, SDK invokes GDB with the STDOUT/STDERR of GDB already redirected to the writing-end of another pipe (pipe1). After processing of the commands, GDB sends the output/confirmation message to the writing-end of pipe1, while SDK reads those feedback messages from the reading-end of pipe1.

The complete pipe structure for the communication between SDK and GDB is presented in Figure 5.3 here:

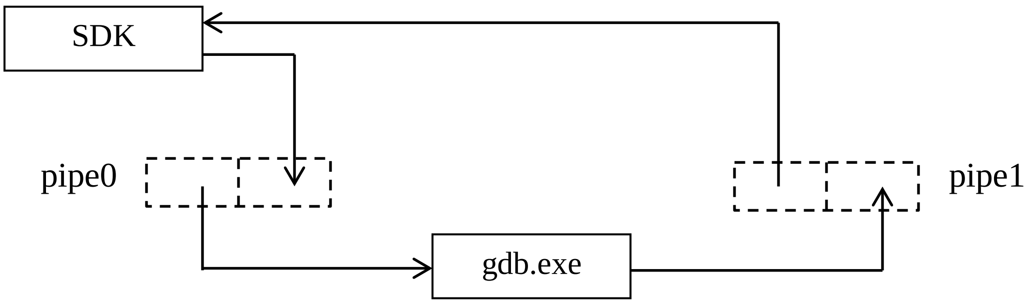


**Figure 5.3:** Redirected-I/O GDB communicates with SDK

Now that the communication model is set and the STDIN handle is grasped, it's time to read data from the reading-end of the pipe, here ReadFile() function is applied for that purpose. Figure 5.4 shows this model:

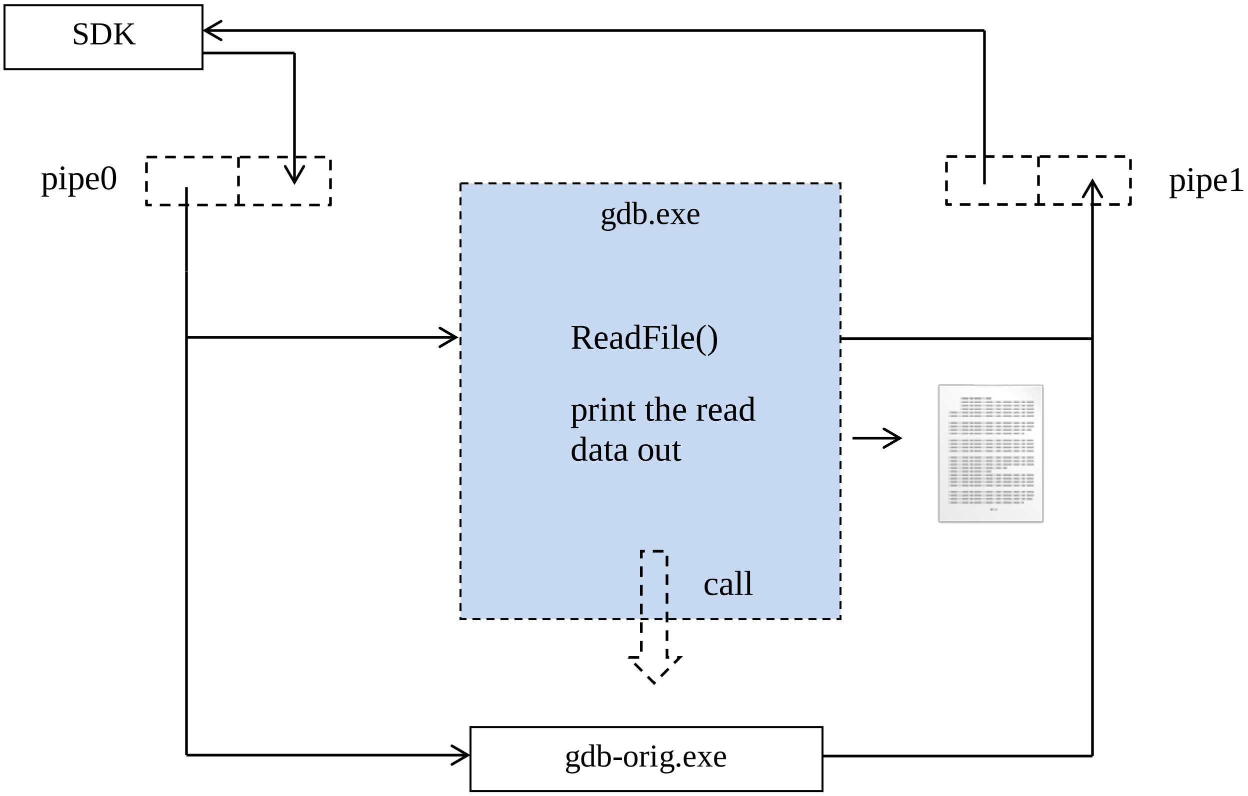


**Figure 5.4:** GDB input stream reading model (i)

In this model, the ReadFile() function, the print-out functionality, as well as the calling of gdb-orig.exe are all implemented within gdb.exe.

Here STDIN of both GDBs are connected to the reading-end of pipe0, and gdb.exe is in charge of calling gdb-orig.exe. We're wishing to read data (commands received from SDK ) from the reading-end of pipe0 and print them out to some file.

And there's no further redirection of STDOUT/STDERR of either gdb.exe or gdb-orig.exe, they are both connected to the writing-end of pipe1.

However, it turns out to be nothing is read, and the debugging session is stucked during the initialization phase. One possible reason is that within gdb-orig.exe a ReadFile() function is also called to read data from its STDIN, which results in a conflict between multiple readers of the same pipe, for example, the pipe can be designed in such a way that once the existence of multiple readers are detected, then

SDK is presented from continuing writing data to the writing-end of the pipe, which causes gdb-orig.exe not to generate the correct confirmation message (as it doesn't receive commands from SDK) and SDK hangs the debugging session.

In order to prevent this potential conflict, an additional pipe and buffer is added here, as Figure 5.5 depicted:

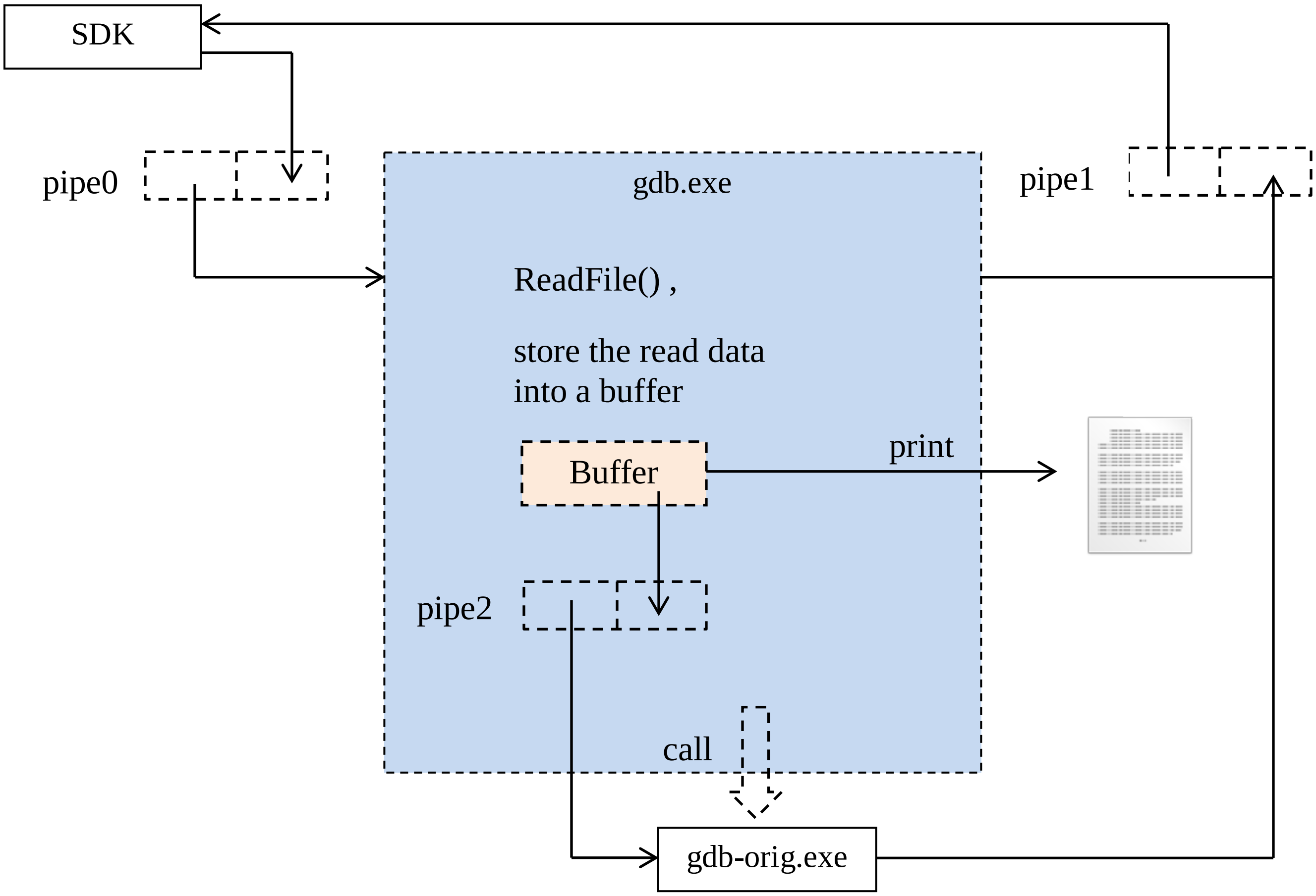


**Figure 5.5:** GDB input stream reading model (ii)

In this model, the ReadFile() function, the data store and print-out functionality, the creation of pipe2, together with the calling of gdb-orig.exe are all implemented within gdb.exe.

Here STDIN of gdb.exe remains unchanged, i.e. still connected to the reading-end of the pipe0. In fact, it is not crucial where the STDIN of gdb.exe is connected to, it can be connected to anywhere else, as long as we specify the ReadFile() function to read the data from the right reading-end of the pipe. However, it is crucial where the STDIN of gdb-orig.exe is connected to, since we're not modifying the source code of gdb-orig.exe, therefore we cannot drive ReadFile() function within gdb-orig.exe to read from nowhere else, but STDIN of it.

STDOUT/STDERR of both gdb.exe and gdb-orig.exe remain unchanged, i.e., still connected to the writing-end of pipe1.

The data flow of this reading model goes as the following: SDK sends the commands to the writing-end of pipe0, the ReadFile() function within gdb.exe reads the commands from the reading-end of pipe0, store them into a temporary buffer for the potential processing later, these commands are printed out to a file, so that we can have a check. At the same time, these commands are forwarded to the writing-end of another pipe, pipe2 which is created earlier, so that gdb-orig.exe can read these commands via the reading-end of pipe2. In this case, STDIN of gdb-orig.exe must be connected to the reading-end of pipe2 for the correct reading.

But adding an intermediate pipe, it's guaranteed that the ReadFile() operations in gdb-orig.exe are later than the ReadFile() operations in gdb.exe. This is ensured by the by the reading/writing principles of pipes: ReadFile() function will not return if the write operation is not completed on the writing-end of the pipe. That is, ReadFile() in gdb-orig.exe will keep on waiting, until some data from the buffer is written to the writing-end of pipe2.

### 5.2.3 Process Calling

As mentioned in the section 5.1, gdb.exe will take the responsibility of calling/executing gdb-orig.exe. Moreover there're some conditions which this calling/executing process must satisfy:

**A)** gdb.exe and gdb-orig.exe must be executed in parallel

i.e. gdb.exe and gdb-orig.exe should both keep executing in parallel until the debugging session is over. gdb.exe needs to keep executing since it has to act as a “fake GDB object” which deceives SDK; while gdb-orig.exe needs to keep executing because it is the actual process who’s reading commands from SDK, processing them, and sending back outputs/confirmation messages.

In order to meet the condition, there’re two more assumptions which must be checked:

**A.1** gdb.exe itself should not end until the debugging session is over, i.e. cannot return from main() function.

**A.2** gdb.exe should not be stucked after calling gdb-orig.exe, i.e. it should not wait for the complete of gdb-orig.exe.

**B)** The standard input (STDIN) of gdb-orig.exe must be redirected

To be exact, STDIN of gdb-orig.exe must be redirected to the reading-end of pipe, as discussed in section 5.2.2.

In order to fulfill the condition A.1, an infinite *for loop* is applied, a piece of pseudo codes are listed in Listing 5.5 to show this idea:

```
for (;;)    //infinite loop

{
// tasks to do
        1. read from reading-end of pipe0
        2. store the read data into a buffer
        3. print the data out to a file
        4. write the data to the writing-end of the pipe2
}
```

**Listing 5.5**: A pseudo-code example with infinite loop applied

By taking advantage of the infinite for loop, it's guaranteed that gdb.exe is keep on repeating the task it's supposed to do and will never end unless the debugging session is over.

As for condition A.2 and condition B), the way that gdb-orig.exe is called is crucial. Here CreateProcess() is chosen to meet these restictions. According to [23], the creation of the new process will not affect the execution of the calling process, which corresponds to condition A.2, on the other side, a STARTUPINFO structure can be specified as the argument of CreateProcess() function, which enables the I/O redirection of the new process. A couple lines of codes are shown in Listing 5.6 to show how it works:

```
// Create pipe2
SECURITY_ATTRIBUTES sa;
sa.bInheritHandle = TRUE;
CreatePipe(&hRead_pipe2,&hWrite_pipe2,&sa,0);

PROCESS_INFORMATION pi;
STARTUPINFO si;

// Set up the STARTUPINFO struct.
ZeroMemory(&si,sizeof(STARTUPINFO));
si.cb = sizeof(STARTUPINFO);
si.dwFlags = STARTF_USESTDHANDLES;

// redirect STDIN of the new process to the reading-end of pipe2
si.hStdInput = hRead_pipe2;

// leave the STDOUT and STDERR undirected
si.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
si.hStdError  = GetStdHandle(STD_ERROR_HANDLE);

// launch the process
CreateProcess(NULL,exe_p,NULL,NULL,TRUE,NULL,NULL,NULL,&si,&pi);
```

**Listing 5.6**: Usage of STARTUPINFO and CreateProcess()

### 5.2.4 Command Cathing Results

By applying the previous mentioned principles and concepts, the commands which SDK sends to GDB during the debugging session can now be captured. Listing 5.7 below shows a small section of them which are recorded (for the complete commands caught, please refer to Appendix A.1):

```
148-gdb-set confirm off
149-gdb-set width 0
150-gdb-set height 0
151-interpreter-exec console echo
152-gdb-show prompt
153-gdb-set auto-solib-add on
154-gdb-set stop-on-solib-events 0
155-gdb-set stop-on-solib-events 1
156-target-select remote localhost:1234
157-target-download
Z:\\Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0\\Debug\\test_proj1_ppc0.elf

…

172-exec-next 1
173 info threads
174-stack-info-depth
175-stack-list-frames 0 1
176-data-list-changed-registers
177 info sharedlibrary
178-stack-list-arguments 0 0 0
179-stack-list-locals 0
180 whatis a
181 whatis b
182 whatis c
183-var-create - * a
184-var-evaluate-expression var1
185-var-create - * b
186-var-evaluate-expression var2
187-var-create - * c
188-var-evaluate-expression var3
189-exec-next 1
190 info threads
191-stack-info-depth
192-stack-list-frames 0 1
193-var-update var1
194-var-update var2
195-var-update var3

…
```

**Listing 5.7**: A section of commands recorded

The listing above clearly shows that the GDB connected to XMD by the command:

156-target-select remote localhost:1234

Where 1234 is the port number that XMD opens.

And from this listing it's also proved that SDK translates the user's actions into a series of GDB commands, as discussed in previous chapter , for instance, *line 172-188* shows the GDB commands generated when user press the "*step ove*r" button for the first time.

## 5.3 Extensions to Dual-processor Debugging

However, the contents discussed in this chapter so far are based on single-processor debugging, i.e., there's only one gdb-orig.exe called by gdb.exe. In order to conform to the semi-auto debugging flow proposed in section 4.3, the extensions to dual-processor scenario should be made.

### 5.3.1 Overview of the Extended Reading Model

In order to perform this extension, three more functions must be augmented:

- One more copy of gdb-orig.exe should be called.
- One more pipe (pipe3) needs to be created.
- Make necessary changes to the data which is read from pipe0, and forward the data to the second gdb-orig.exe via writing the data into the writing-end of pipe3.

In below, the reading model for dual-processor scenario is shown in Figure 5.6:

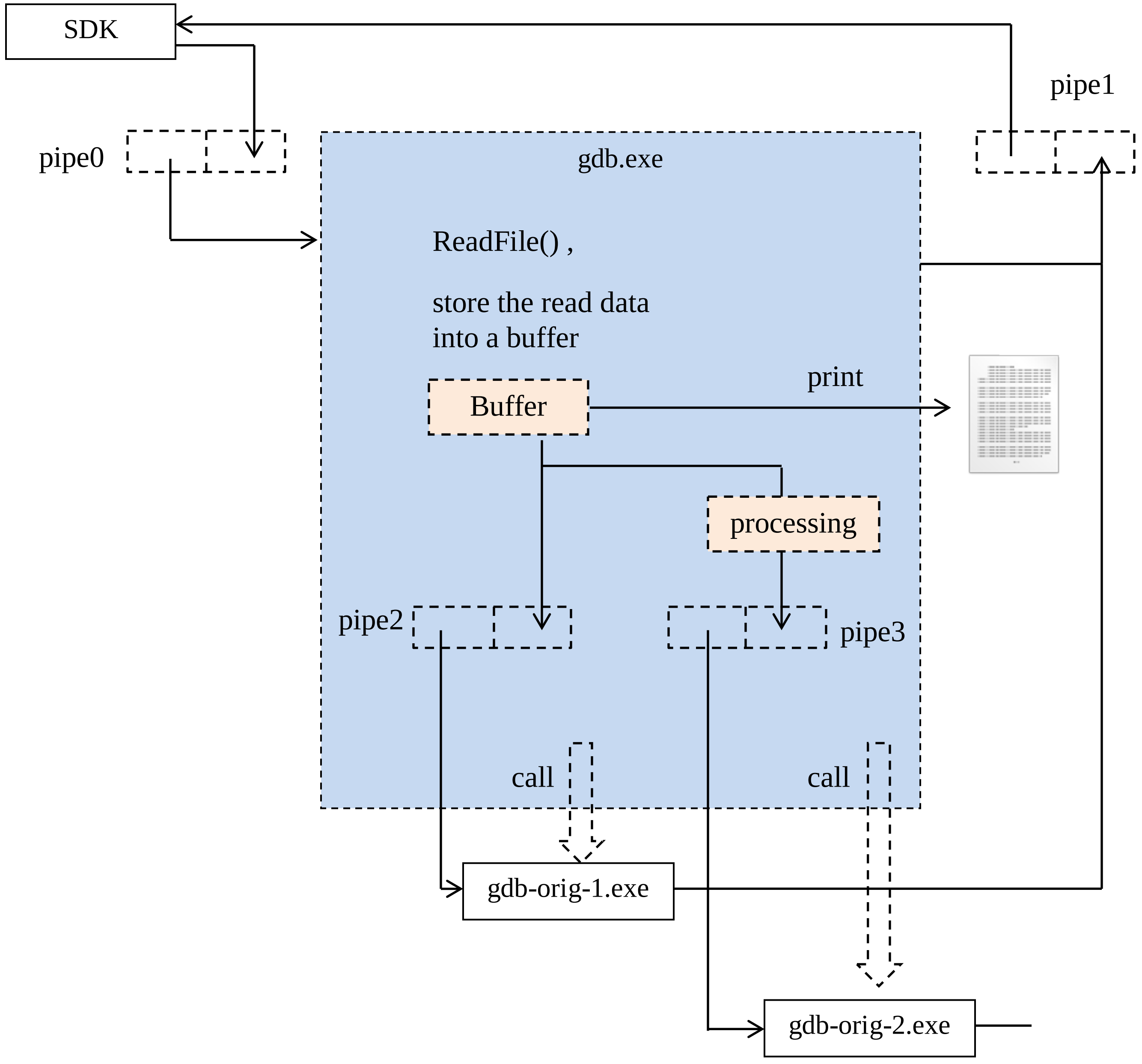


**Figure 5.6:** GDB input stream reading model for dual-processor debugging

In this model, the original GDB is copied and renamed as gdb-orig-1.exe and gdb-orig-2.exe respectively, while in debugging process, the three executables are running in parallel, and will not end before the debugging session is over.

Besides that, two more points should be noticed:

1. In the “processing” block, all the messages are copied from the buffer, except the connecting-to-XMD command.
2. The STDOUT/STDERR of gdb-orig-2.exe should not be connected to the writing-end of pipe1.

The detailed discussion and explanation of these two points will be presented in the following two subsections.

### 5.3.2 Command Processing Block

From the general reading model, the commands that SDK sends to GDB are duplicated as two copies, which are forwarded to pipe2 and pipe3 respectively. However, there’s an exception: the connecting-to-XMD command, i.e. the following command:

156-target-select remote localhost:1234

As mentioned in section 4.3, in semi-auto dual-processor debugging scenario, two XMD instances will be used to connect to both PowerPC targets individually, and therefore two GDB servers are opened with different port numbers, thus, the port number here(1234) must be modified to be different. For the automatic connection path in Figure 4.12, SDK will get the port number from XMD automatically and use it to generate GDB commands.  However, in the manual connection path, the correct port number is only known after the connection is set, then the C source codes of gdb.exe must be adjusted accordingly to the correct port number, and then the compiled gdb.exe can be copied to overwrite the formal one, lastly the debugging session can be started. Figure 5.7 shows this processing flow:

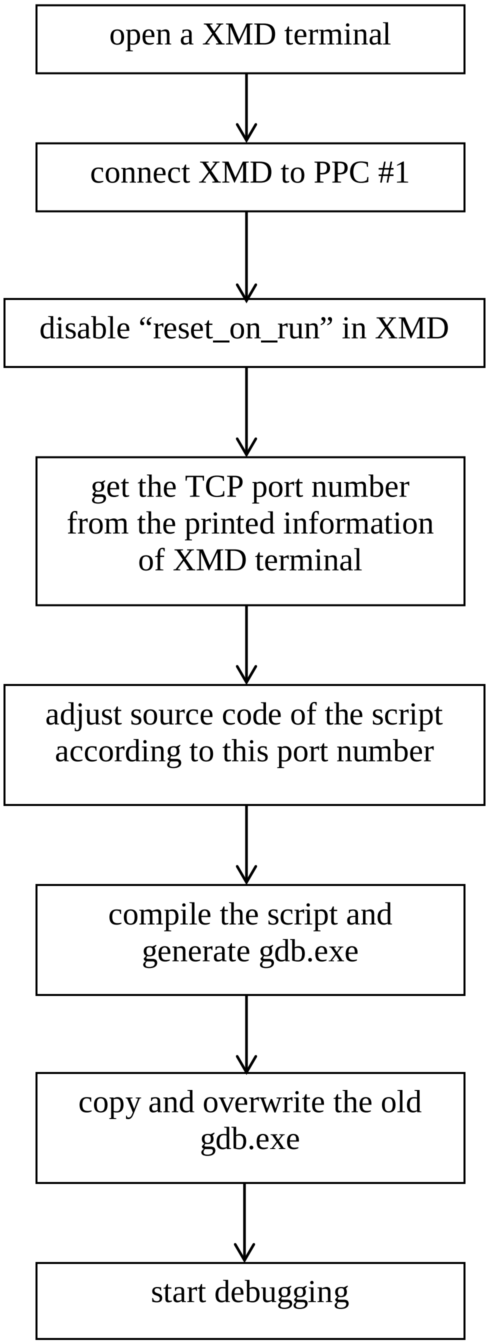


**Figure 5.7:** The processing flow before starting debugging session

And a small piece of C codes to do the XMD port modification work (i.e. the “processing” block in Figure 5.6), is shown in Listing 5.8:

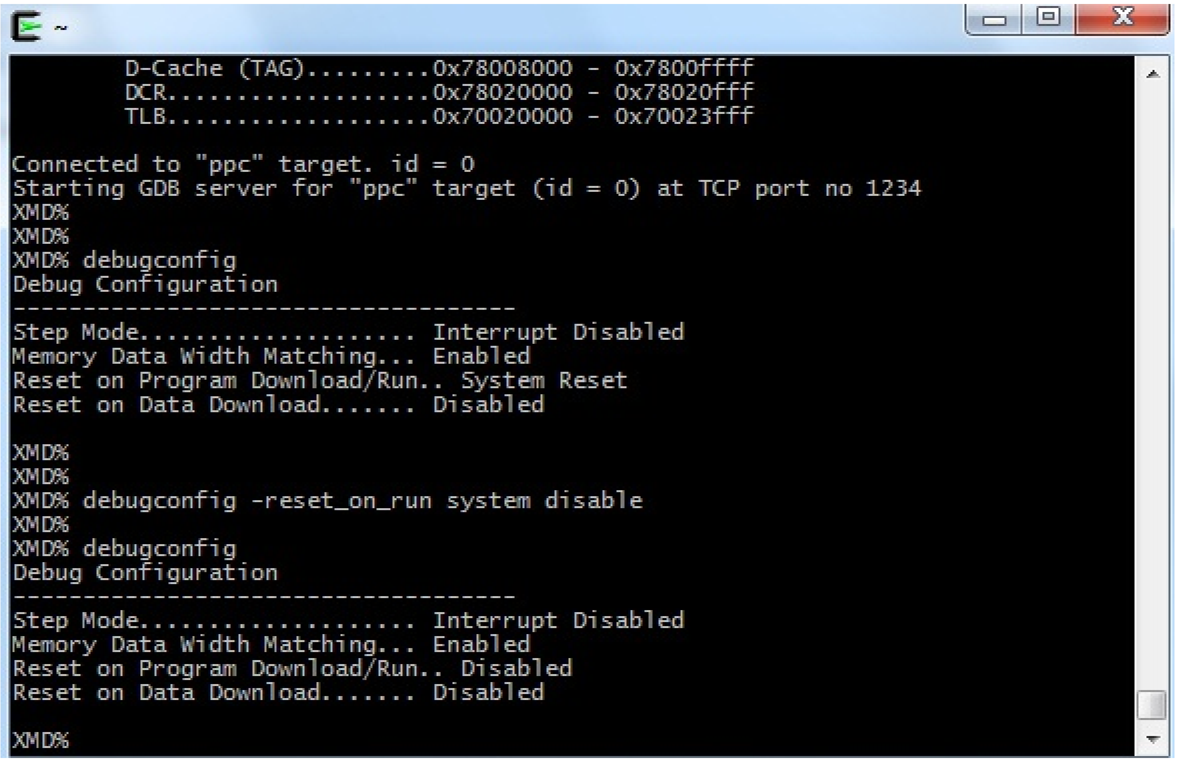
```
#define XMD_PORT "1234"

// copy the obtained commands into buffer
memcpy(cmd_1,stdin_buf,dwRead);

// if the connect-to-XMD command is found
if((str_fnd = strstr(stdin_buf,”remote localhost”)) != NULL)

// substitute the port number with XMD_PORT
        memcpy(&cmd_1[strlen(cmd_1) -1-  strlen(XMD_PORT)],
                    XMD_PORT,strlen(XMD_PORT));
```

**Listing 5.8:** C implementation of XMD port modification

As previously stated, the value of XMD_PORT might be re-defined, according the port number captured after the manual XMD-PowerPC connection.

In Figure 5.7, the step “disable reset_on_run in XMD” is important, otherwise the whole system will be reset (by default) once the debugging session is started, which will result the automatic debugging process to be suspended and stucked.

Figure 5.8 shows the information printed in XMD terminal, with the port number included, as well as the system debugconfig information before and after “disable reset_on_run” is applied.



```
        D-Cache (TAG).........0x78008000 - 0x7800ffff
        DCR...................0x78020000 - 0x78020fff
        TLB...................0x70020000 - 0x70023fff

Connected to "ppc" target. id = 0
Starting GDB server for "ppc" target (id = 0) at TCP port no 1234
XMD%
XMD%
XMD% debugconfig
Debug Configuration
-----------------------------------
Step Mode.................... Interrupt Disabled
Memory Data Width Matching... Enabled
Reset on Program Download/Run.. System Reset
Reset on Data Download....... Disabled

XMD%
XMD%
XMD% debugconfig -reset_on_run system disable
XMD%
XMD% debugconfig
Debug Configuration
-----------------------------------
Step Mode.................... Interrupt Disabled
Memory Data Width Matching... Enabled
Reset on Program Download/Run.. Disabled
Reset on Data Download....... Disabled

XMD%
```

**Figure 5.8:** Information printed in XMD terminal

### 5.3.3 Connection of STDOUT/STDERR

As shown in Figure 5.6, the STDOUT/STDERR of gdb.exe and gdb-orig-1.exe are connected to the writing-end of pipe1, while the STDOUT/STDERR of gdb-orig-2.exe should not be connected.

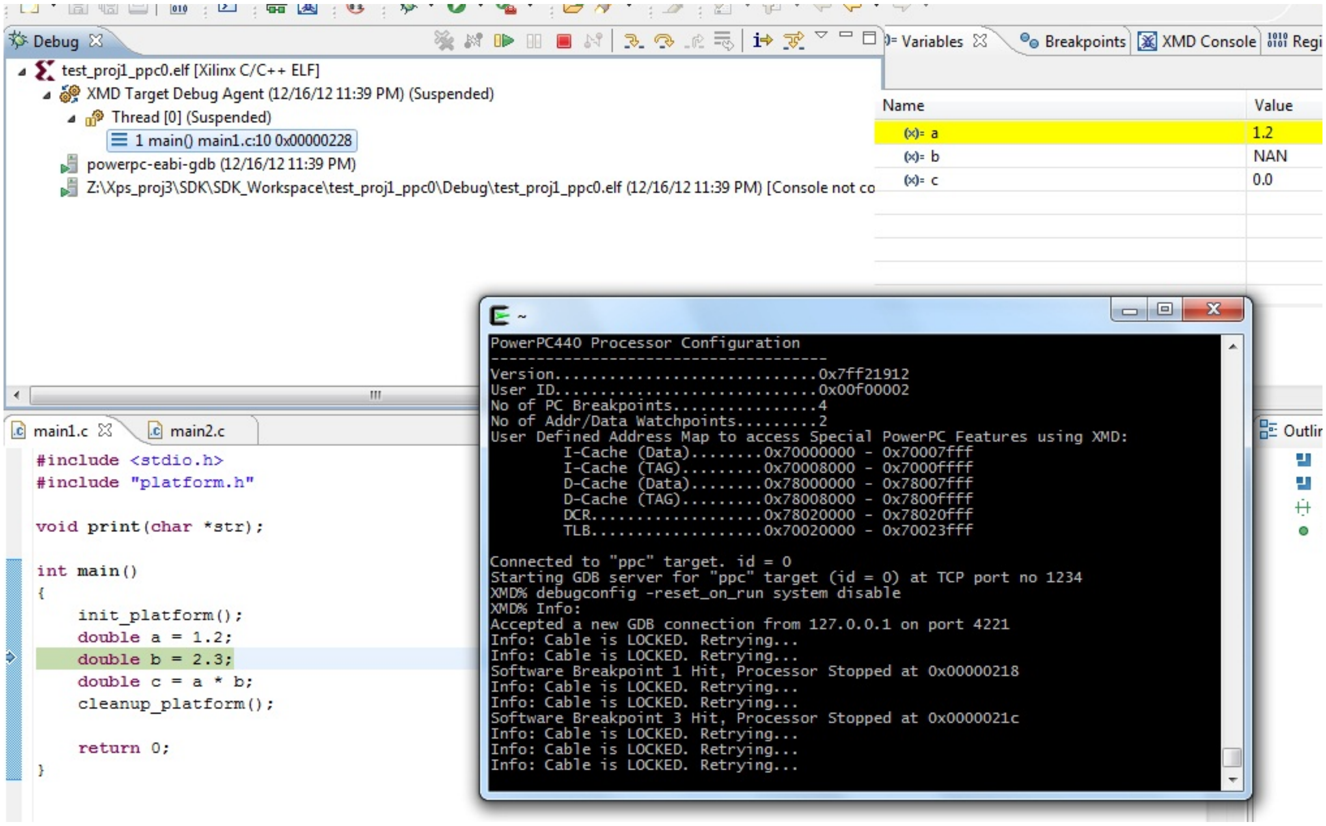
The reason is gdb-orig-1.exe already passes the output/confirmation messages to SDK via writing these messages to the writing-end of pipe1, if the STDOUT/STDERR of gdb-orig-2.exe is again connected to the writing-end of pipe1, then SDK will receive two copies of the feedback messages. However, SDK is defined to be able to process one copy only at a time, thus it hangs if it receives two copies and the debugging session cannot be continued.

As a conclusion, the STDOUT/STDERR of gdb-orig-2.exe can be redirecte to anywhere else, except the writing-end of pipe1.

## 5.4 Output Catching/Forwarding

### 5.4.1 GDB Output Stream Writing Model

By taking advantage of the functionalities which are implemented up to now, the user can debug the same piece of code synchronously in both PowerPC processors. Figure 5.9 shows the screenshot of semi-auto dual-processor debugging session:



```
test_proj1_ppc0.elf [Xilinx C/C++ ELF]
  XMD Target Debug Agent (12/16/12 11:39 PM) (Suspended)
    Thread [0] (Suspended)
      1 main() main1.c:10 0x00000228
  powerpc-eabi-gdb (12/16/12 11:39 PM)
  Z:\Xps_proj3\SDK\SDK_Workspace\test_proj1_ppc0\Debug\test_proj1_ppc0.elf (12/16/12 11:39 PM) [Console not co
```

| Name | Value |
|---|---|
| (x)= a | 1.2 |
| (x)= b | NAN |
| (x)= c | 0.0 |

```c
#include <stdio.h>
#include "platform.h"

void print(char *str);

int main()
{
    init_platform();
    double a = 1.2;
    double b = 2.3;
    double c = a * b;
    cleanup_platform();

    return 0;
}
```

```
PowerPC440 Processor Configuration
-------------------------------------
Version.............................0x7ff21912
User ID.............................0x00f00002
No of PC Breakpoints................4
No of Addr/Data Watchpoints.........2
User Defined Address Map to access Special PowerPC Features using XMD:
        I-Cache (Data)........0x70000000 - 0x70007fff
        I-Cache (TAG).........0x70008000 - 0x7000ffff
        D-Cache (Data)........0x78000000 - 0x78007fff
        D-Cache (TAG).........0x78008000 - 0x7800ffff
        DCR...................0x78020000 - 0x78020fff
        TLB...................0x70020000 - 0x70023fff

Connected to "ppc" target. id = 0
Starting GDB server for "ppc" target (id = 0) at TCP port no 1234
XMD% debugconfig -reset_on_run system disable
XMD% Info:
Accepted a new GDB connection from 127.0.0.1 on port 4221
Info: Cable is LOCKED. Retrying...
Info: Cable is LOCKED. Retrying...
Software Breakpoint 1 Hit, Processor Stopped at 0x00000218
Info: Cable is LOCKED. Retrying...
Info: Cable is LOCKED. Retrying...
Software Breakpoint 3 Hit, Processor Stopped at 0x0000021c
Info: Cable is LOCKED. Retrying...
Info: Cable is LOCKED. Retrying...
Info: Cable is LOCKED. Retrying...
```

**Figure 5.9:** Screenshot of semi-auto dual-processor debugging session

The screenshot clearly shows that the SDK auto-debugging path goes smoothly, and the manual debugging path is proven to be also in progress by the printed information in XMD terminal like “Accepting GDB connection”, “Software breakpoints hit” etc.

However, it would be more convinced if the output of the GDBs can be examined, and when a test project is applied on the hardware platform which supports DSA (in Chapter 3) with the random rounding mode applied, and different values of the same variable can be collected and presented to users, which is also one of the pre-set targets.

As discussed in 5.2.2, the pipe structure is assumed for the communications between SDK and GDB. Similar to Figure 5.5, a GDB output stream writing model with additional buffer and pipe applied, is proposed here in Figure 5.10 below:

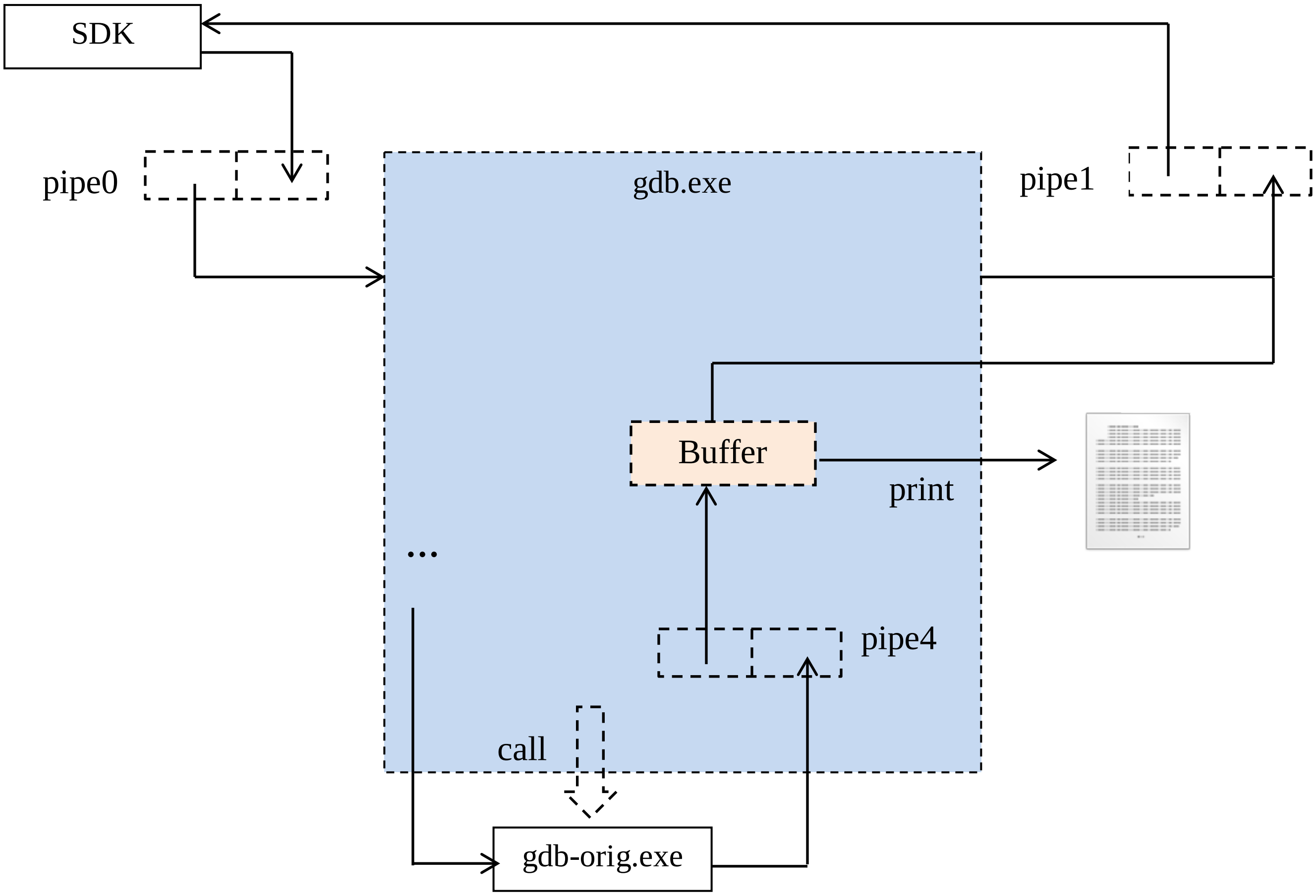


**Figure 5.10:** GDB Output Stream Writing Model

Here the connection of STDIN of gdb-orig.exe is omitted due to the space limitation, and only the STDOUT/STDERR of it should be focused.

The STDOUT/STDERR of gdb-orig.exe is redirected to a writing-end of pipe4, which is created earlier within gdb.exe. Again ReadFile() function is applied to read data from the reading-end of pipe4, and store them into a temporary buffer, after the contents in buffer are printed out to an external file, these contents are also written to the writing-en of pipe1, so that SDK can read them from the corresponding reading end.

A corresponding pseudo code is shown in Listing 5.9:

```
// Set all the necessary connections/redirections

for (;;)    //infinite loop

{

    1. perform input stream command catching (same source codes as in section 5.2)
    2. perform output stream command catching

}
```

**Listing 5.9**: A pseudo-code of output catching implementation

By this means we’re wishing to catch the output of gdb-orig.exe without affecting the debugging process. However, it would be not be successful, and SDK would generate an error when entering debug perspective.

It’s due to the blocking behavior caused by the reading/writing principles of pipes: the ReadFile() /WriteFile() function will only return, if the number of required bytes has been read/written, or a write/read operation is completed in the writing-end/reading-end of the pipe, respectively. Otherwise, it will keep waiting until the write/read operation is done.

Therefore, there’s no problem, when we implement the input stream command catching in the infinite loop: the ReadFile() function in gdb.exe will wait until some data is written to pipe0, and a carriage return is virtually hit. Then the ReadFile() function will start to read and in this way the data in pipe is flowing.

However, when the same procedure for the GDB output catching is implemented in the same *for loop*, the ReadFile() function which is supposed to read the output of GDB, is waiting for GDB to write its output to the writing-end of pipe4. But the input command and the output message is not a one-by-one correspondence, which means, when GDB receives a command from SDK, it doesn't necessarily generate one output message, there’re also situations that GDB will generate one output message only when two or more commands are obtained.

In this situation, the ReadFile() function is waiting for the output message, which GDB will never generate until it receives the next command from SDK, but the “receiving” process can only be done in the next loop run. Therefore the program is stucked in this point.

The solution is running the input command catching codes and the output message catching codes in two separate loops, and the two loops must be executing in parallel.

This is done by creating separate threads for both *for-loops* in main() function, and waiting for the finish of both threads for infinite time

A piece of pseudo code in Listing 5.10 shows the idea:

```
// Thread to do input command catching
DWORD WINAPI Thread_1(void* pVoid)
{
for (;;)    //infinite loop
          {
                    do input command catching;
          }

}

// Thread to do output message catching
DWORD WINAPI Thread_2(void* pVoid)
{
for (;;)    //infinite loop
          {
                    do output message catching;
          }

}

main()
{
//create both threads
          CreateThread1;
          CreateThread2;

//waiting for both threads to finish, for infinite time
WaitForMultipleObjects(2, hThread, TRUE, INFINITE);

}
```

**Listing 5.10**: A pseudo-code of input/output catching implementation

Extension to dual-processor debugging scenario is also very similar as described in section 5.3. Instead of showing the structure for output catching only, a complete diagram for both input and output catching implementation is shown below in Figure 5.11:

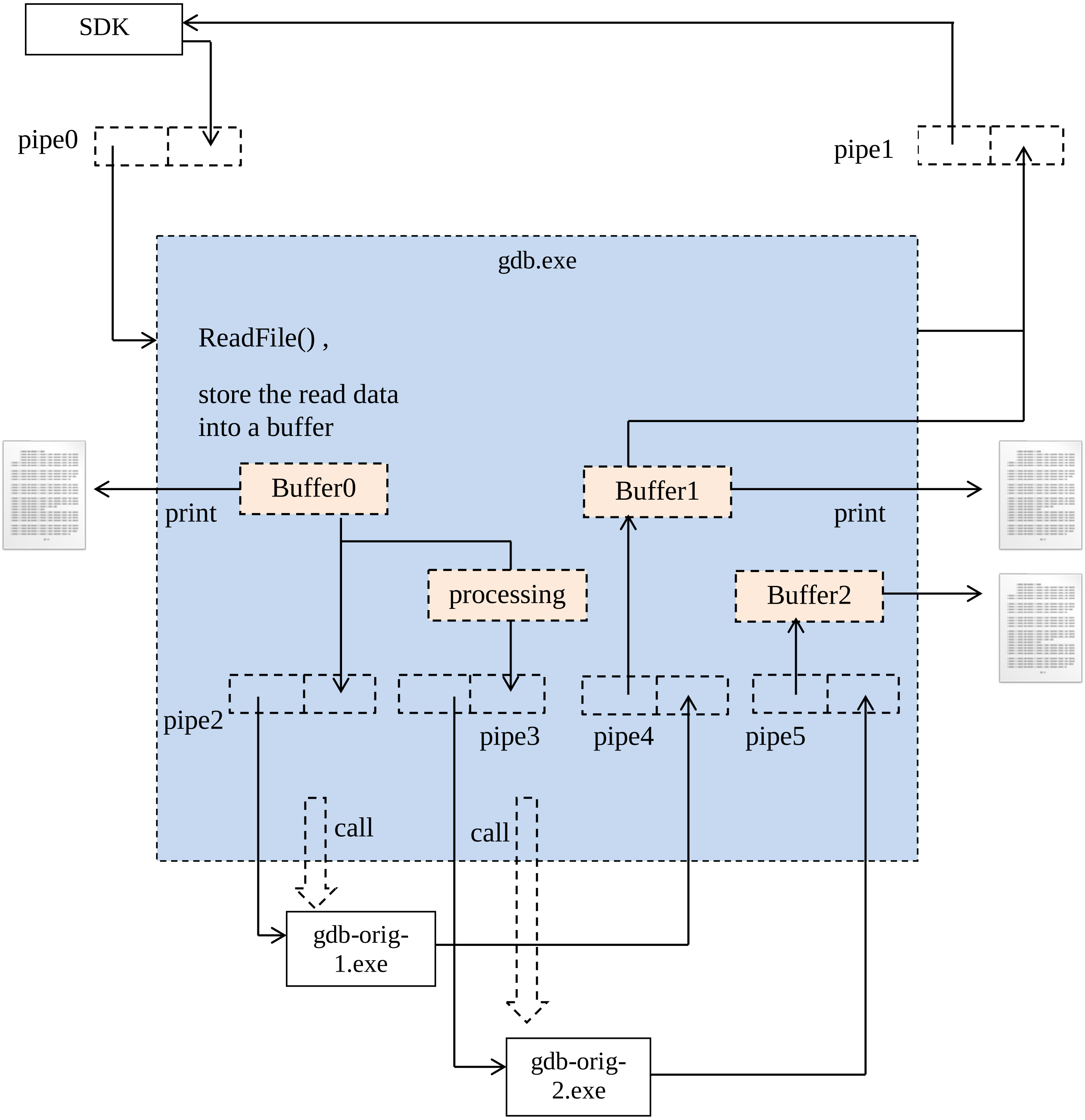


**Figure 5.11**: A complete diagram about the input/output catching implementation

Here Buffer2 should not be connected to the writing-end of pipe1, according to the discussion in section 5.3.3. In addition, the input commands catching(for both GDBs), the output message catching for gdb-orig-1, the output message catching for gdb-orig-2, should be included in three threads respectively.

### 5.4.2 Output Message Catching Results

Listing 5.11 shows a part of the output message catching result for one GDB, for the complete catching results, please refer to Appendix A.2.

```
123^done,changelist=[]
(gdb)
124^done,changelist=[]
(gdb)
125^done,changelist=[{name="var3",in_scope="true",type_changed="false"}]
(gdb)
126^done,changed-registers=["32","45","64","70","114","174"]
(gdb)
&"info sharedlibrary\n"
~"No shared libraries loaded at this time.\n"
127^done
(gdb)
128^done,stack-args=[frame={level="0",args=[]}]
(gdb)
129^done,locals=[name="a",name="b",name="c"]
(gdb)
130^done,value="2.7599999999999998"
(gdb)
131^done,value="2.7599999999999998"
(gdb)
132^running
(gdb)
132*stopped,reason="end-stepping-range",thread-
id="0",frame={addr="0x00000248",func="main",args=[],file="../src/main1.c",fullname="Z:\\Xps
_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="14"}
(gdb)
```

**Listing 5.11**: A section of output messages recorded

In this listing, it's proved that GDB generates some output/confirmation messages as a respond to the commands received. These messages are read and processed by SDK, some of them are recognize as the confirmation message (e.g. ^done ), while some of them are taken as the required variable value, which will be displayed in the SDK's graphical debugging interface (e.g. value="2.7599999999999998").

## 5.5 Results Collection and Calculation of Precision

Now the actual hardware architecture which is described in Chapter 3 is applied, with a new software application for testing our numerical accuracy debugger. Listing 5.12 shows the main source codes of the testing application:

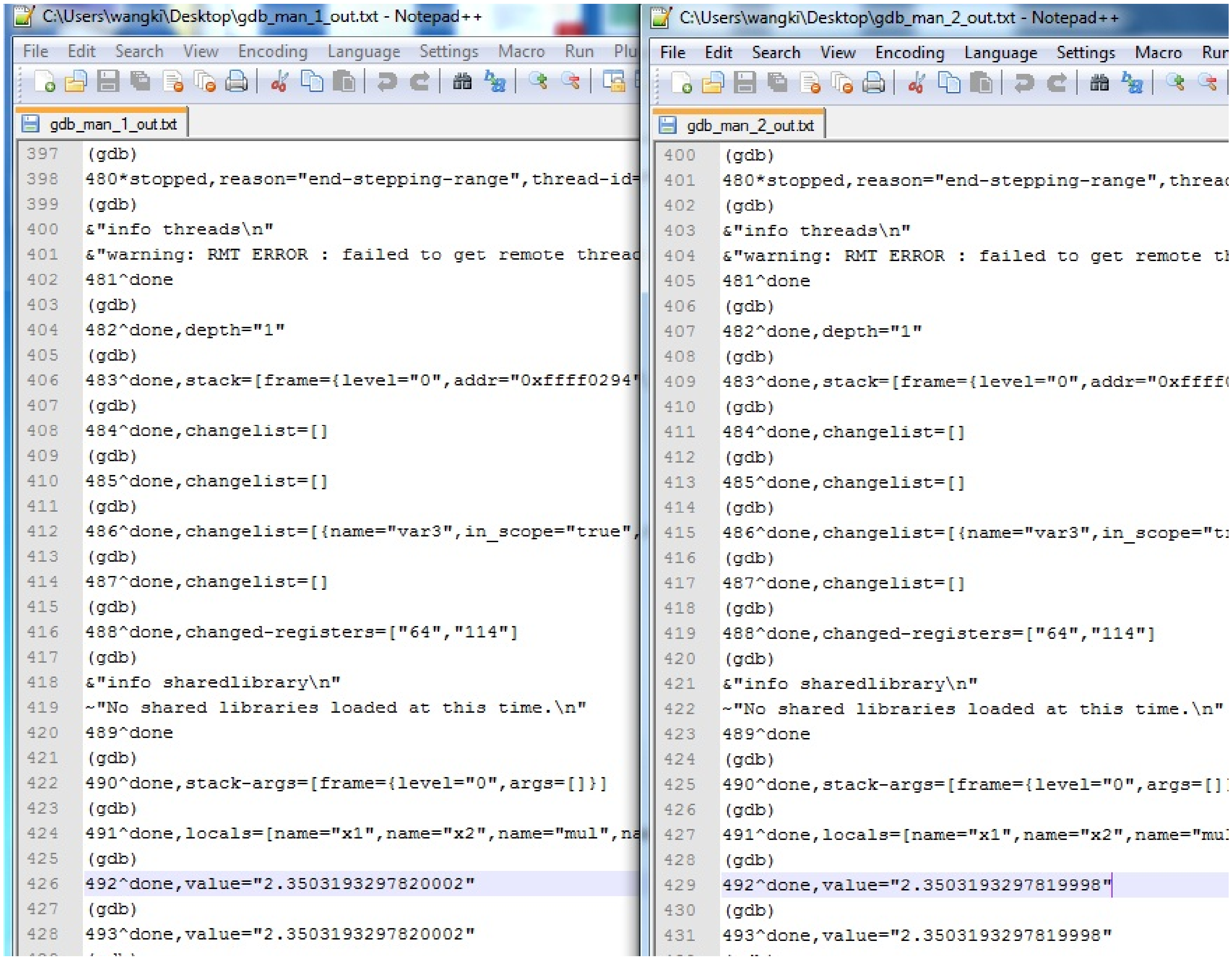
```
double x1 = 1.791234;
double x2 = 1.312123;
double mul;

int i=0;

for( i=0; i<10; i++ )
{
        mul = x1*x2;

}
```

**Listing 5.12**: C codes of the test software application

Here the same floating point multiplication is performed for 10 times, to make sure that different results can be obtained with random rounding mode applied.

By debugging this software synchronously in both PowerPC processors based on the hardware system with DSA support, two output .txt files can be obtained, which records the output messages of both GDBs respectively.

By examing these two files, all the output messages are identical, except the value of variable mul. It is shown in Figure 5.12 that different values of mul are obtained from different PowerPC processors:



```
(gdb)
480*stopped,reason="end-stepping-range",thread-id=
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote threa
481^done
(gdb)
482^done,depth="1"
(gdb)
483^done,stack=[frame={level="0",addr="0xffff0294"
(gdb)
484^done,changelist=[]
(gdb)
485^done,changelist=[]
(gdb)
486^done,changelist=[{name="var3",in_scope="true",
(gdb)
487^done,changelist=[]
(gdb)
488^done,changed-registers=["64","114"]
(gdb)
&"info sharedlibrary\n"
~"No shared libraries loaded at this time.\n"
489^done
(gdb)
490^done,stack-args=[frame={level="0",args=[]}]
(gdb)
491^done,locals=[name="x1",name="x2",name="mul",na
(gdb)
492^done,value="2.3503193297820002"
(gdb)
493^done,value="2.3503193297820002"
```

```
(gdb)
480*stopped,reason="end-stepping-range",threa
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote t
481^done
(gdb)
482^done,depth="1"
(gdb)
483^done,stack=[frame={level="0",addr="0xffff
(gdb)
484^done,changelist=[]
(gdb)
485^done,changelist=[]
(gdb)
486^done,changelist=[{name="var3",in_scope="t
(gdb)
487^done,changelist=[]
(gdb)
488^done,changed-registers=["64","114"]
(gdb)
&"info sharedlibrary\n"
~"No shared libraries loaded at this time.\n"
489^done
(gdb)
490^done,stack-args=[frame={level="0",args=[]
(gdb)
491^done,locals=[name="x1",name="x2",name="mu
(gdb)
492^done,value="2.3503193297819998"
(gdb)
493^done,value="2.3503193297819998"
```

**Figure 5.12**: Different values of mul from different processors

From this test, it’s proven that the numerical accuracy debugger works perfectly and the results can be collected by checking for the output messages of GDBs. As long as the results are gathered, the number of significant bits can be straightly calculated, according to the equation presented in section 2.3.1.

The implementation is also not complicated. From Figure 5.11, a Collecting/Computing block is added on top of buffer #1 and buffer #2, this block is in charge of collecting both values of the same variable by examing the output messages of both GDBs which are stored in the buffer. The block then extracts both values , calculates the number of significant bits and returns it to SDK .

As discussed in section 5.4.2, SDK will take use of the output messages of GDB, extract the values of variables and display them on the graphical debugging interface. Therefore, after both values are collected and the number of significant bits are estimated, the Collecting/Computing block will modify the values of the variable (e.g. value="2.7599999999999998” will be modified) to the following format shown in Figure 5.13:

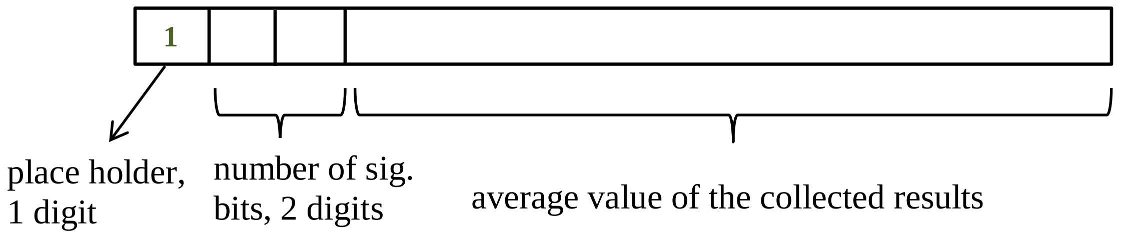
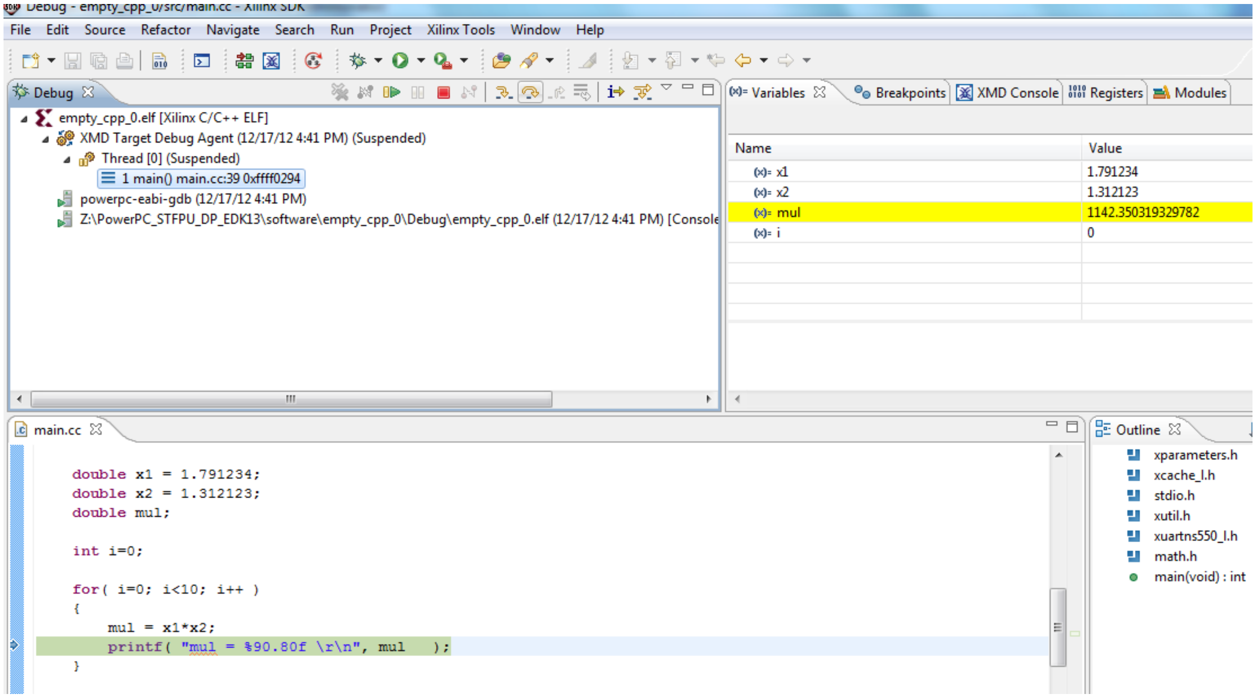


**Figure 5.13**: Customized format for SDK reading

Figure 5.14 shows the actual screenshot of SDK reading the modified value and display it in the graphical debug perspective.



```
empty_cpp_0.elf [Xilinx C/C++ ELF]
  XMD Target Debug Agent (12/17/12 4:41 PM) (Suspended)
    Thread [0] (Suspended)
      1 main() main.cc:39 0xffff0294
  powerpc-eabi-gdb (12/17/12 4:41 PM)
  Z:\PowerPC_STFPU_DP_EDK13\software\empty_cpp_0\Debug\empty_cpp_0.elf (12/17/12 4:41 PM) [Console
```

| Name | Value |
|---|---|
| x1 | 1.791234 |
| x2 | 1.312123 |
| mul | 1142.350319329782 |
| i | 0 |

```
    double x1 = 1.791234;
    double x2 = 1.312123;
    double mul;

    int i=0;

    for( i=0; i<10; i++ )
    {
        mul = x1*x2;
        printf( "mul = %90.80f \r\n", mul    );
    }
```

```
xparameters.h
xcache_l.h
stdio.h
xutil.h
xuartns550_l.h
math.h
main(void) : int
```

**Figure 5.14**: SDK reads the modified value and displays it

Lastly, Figure 5.15 shows the final diagram of the graphical numerical accuracy debugger (the print-out lines are omitted).

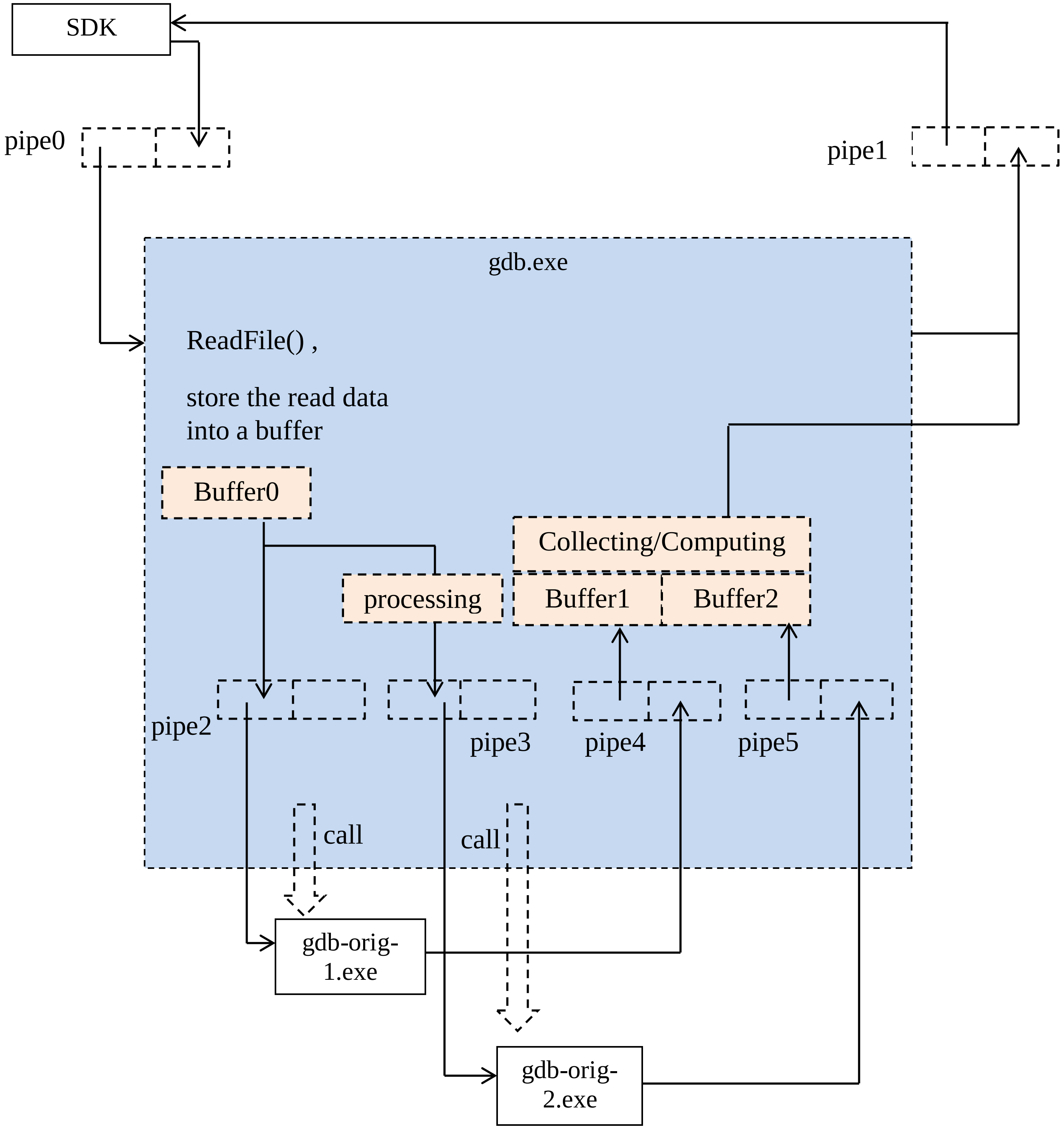


**Figure 5.15**: Final diagram of the graphical numerical accuracy debugger

# 6 Conclusions and Future Work

In this work, a graphical numerical accuracy debugger based on an FPGA computing system is developped. Using this debugger, without source code modification, the user's program can be executed with random rounding on the N parallel processing blocks of the FPGA based computing system, and numerical accuracy information of any variable can be generated according to the Discrete Stochastic Arithmetic (DSA) and reported to the user.

Starting from the investigation of Xilinx SDK debugging flow, a semi-auto dual-processor debugging flow is proposed. In this debugging flow, a manual GDB-XMD-PowerPC connection is set in parallel with the SDK‘s automatic debugging path, so that when an executable file is required to be debugged on hardware, the commands which are sent from SDK to GDB can be captured, processed and forwarded to another GDB instance, which realizes the functionality of synchronously debugging.

The implementation of the proposed debugging flow is done via substituting the original GDB by a script. Within the script, functionalities like the input commands catching, processing, output messages catching etc, are implemented. In addition, by checking the output messages of GDBs, different values of the same variable can be extracted and used as the calculation of number of significant bits. Afterwards, the obtained accuracy, as well as the computed results can be displayed in SDK‘s graphical debugging interface by replacing the values in GDB output message with a customized format.

## Future Work

In the current graphical numerical accuracy debugger implementation, the number of significant digits and the computed results can only be displayed in SDK graphical debugging interface, via the method of examing the output messages of GDB and replacing the actual values with a pre-defined format, as shown in section 5.5. By investigating the eclipse plug-ins, it would be possible to display the number of significant digits and both random rounding results from both processors in a more general and user-friendly way.

Moreover, according to the hardware platform specification, it is possible for PowerPC processors to catch the NAU exception which is raised whenever any kind of numerical instability is detected. Through reading the value of corresponding registers, the syndrome (catagories of numerical instabilities) can be located. Thus it would be very helpful if this syndrome can be displayed in SDK's graphical debugging interface, so that the user can have a direct view of the types of the numerical instabilities detected.

# A Appendix

## A.1 Complete Commands Catched During a Debugging session

```
79-gdb-set confirm off
80-gdb-set width 0
81-gdb-set height 0
82-interpreter-exec console echo
83-gdb-show prompt
84-gdb-set auto-solib-add on
85-gdb-set stop-on-solib-events 0
86-gdb-set stop-on-solib-events 1
87-target-select remote localhost:1241
88-target-download Z:\\Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0\\Debug\\test_proj1_ppc0.elf
89-environment-directory Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0/Debug
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0/Debug/src
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0/src
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/code
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/include
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/lib
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/apu_fpu_virtex5_v1_00_a
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/apu_fpu_virtex5_v1_00_a/src
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/bram_v3_01_a
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/bram_v3_01_a/src
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/common_v1_00_a
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/common_v1_00_a/src
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/cpu_ppc440_v2_01_a
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/cpu_ppc440_v2_01_a/src
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/lldma_v2_00_a
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/lldma_v2_00_a/src
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/memcon_v2_00_a
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/memcon_v2_00_a/src
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/standalone_v3_07_a
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/standalone_v3_07_a/src
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/standalone_v3_07_a/src/profile
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/uartlite_v2_00_a
Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/uartlite_v2_00_a/src
Z:/Xps_proj3/SDK/SDK_Workspace/Xps_proj3_hw_platform
Z:/Xps_proj3/SDK/SDK_Workspace/Xps_proj3_hw_platform/cache
Z:/Xps_proj3/SDK/SDK_Workspace/Xps_proj3_hw_platform/settings Z:/
90 info threads
91-data-list-register-names
92-break-insert -t exit
```

```
93-stack-info-depth
94-stack-list-frames 0 1
95-break-insert -t main
96-exec-continue
97 info threads
98-stack-info-depth
99-stack-list-frames 0 1
100-data-list-changed-registers
101 info sharedlibrary
102-stack-list-arguments 0 0 0
103-stack-list-locals 0
104 whatis a
105 whatis b
106 whatis c
107-var-create - * a
108-var-evaluate-expression var1
109-var-create - * b
110-var-evaluate-expression var1
111-var-create - * c
112-var-evaluate-expression var2
113-var-evaluate-expression var2
114-var-evaluate-expression var3
115-var-evaluate-expression var3
116-exec-next 1
117 info threads
118-stack-info-depth
119-stack-list-frames 0 1
120-var-update var1
121-var-update var2
122-var-update var3
123-data-list-changed-registers
124 info sharedlibrary
125-stack-list-arguments 0 0 0
126-stack-list-locals 0
127-exec-next 1
128 info threads
129-stack-info-depth
130-stack-list-frames 0 1
131-var-update var1
132-var-update var2
133-var-update var3
134-data-list-changed-registers
135 info sharedlibrary
136-stack-list-arguments 0 0 0
137-stack-list-locals 0
138-var-evaluate-expression var1
139-var-evaluate-expression var1
140-exec-next 1
141 info threads
142-stack-info-depth
143-stack-list-frames 0 1
144-var-update var1
```

```
145-var-update var2
146-var-update var3
147-data-list-changed-registers
148 info sharedlibrary
149-stack-list-arguments 0 0 0
150-stack-list-locals 0
151-var-evaluate-expression var2
152-var-evaluate-expression var2
153-exec-next 1
154 info threads
155-stack-info-depth
156-stack-list-frames 0 1
157-var-update var1
158-var-update var2
159-var-update var3
160-data-list-changed-registers
161 info sharedlibrary
162-stack-list-arguments 0 0 0
163-stack-list-locals 0
164-var-evaluate-expression var3
165-var-evaluate-expression var3
166-exec-next 1
167 info threads
168-stack-info-depth
169-stack-list-frames 0 1
170-var-update var1
171-var-update var2
172-var-update var3
173-data-list-changed-registers
174 info sharedlibrary
175-stack-list-arguments 0 0 0
176-stack-list-locals 0
177-exec-next 1
178 info threads
179-stack-info-depth
180-stack-list-frames 0 1
181-var-update var1
182-var-update var2
183-var-update var3
184-data-list-changed-registers
185 info sharedlibrary
186-stack-list-arguments 0 0 0
187-stack-list-locals 0
188-exec-next 1
189 info threads
190-stack-info-depth
191-stack-info-depth
192-stack-list-frames 0 2
193-var-update var1
194-var-update var2
195-var-update var3
196-data-list-changed-registers
```

```
197 info sharedlibrary
198-stack-list-arguments 0 0 0
199-stack-list-locals 0
200 kill
201-gdb-exit
```

## A.2 Complete Output (for 1 GDB) Catched During a Debugging session

```
&".gdbinit: No such file or directory.\n"
(gdb)
45^done
(gdb)
46^done
(gdb)
47^done
(gdb)
48^done
(gdb)
49^done,value="(gdb) "
(gdb)
50^done
(gdb)
51^done
(gdb)
52^done
(gdb)
Connected to a PPC440 target.
53^connected,thread-
id="0",frame={addr="0x00000218",func="main",args=[],file="../src/main1.c",fullname="Z:\\Xps_proj3\
\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="8"}
(gdb)
54+download,{section=".text",section-size="2352",total-size="42981"}
54+download,{section=".text",section-sent="2352",section-size="2352",total-sent="2352",total-
size="42981"}
54+download,{section=".init",section-size="36",total-size="42981"}
54+download,{section=".fini",section-size="32",total-size="42981"}
54+download,{section=".rodata",section-size="18",total-size="42981"}
54+download,{section=".data",section-size="248",total-size="42981"}
54+download,{section=".got2",section-size="28",total-size="42981"}
54+download,{section=".ctors",section-size="8",total-size="42981"}
54+download,{section=".dtors",section-size="8",total-size="42981"}
54+download,{section=".eh_frame",section-size="8",total-size="42981"}
54+download,{section=".jcr",section-size="4",total-size="42981"}
54+download,{section=".sdata",section-size="8",total-size="42981"}
54+download,{section=".boot0",section-size="204",total-size="42981"}
54+download,{section=".boot",section-size="4",total-size="42981"}
54^done,address="0xfffffffc",load-size="2958",transfer-rate="94656",write-rate="227"
(gdb)
```

```
55^done,source-
path="Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc
0/Debug:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0/Debug/src:Z:/Xps_proj3/SDK/SDK_Workspace/te
st_proj1_ppc0/src:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp:Z:/Xps_proj3/SDK/SDK_Workspac
e/test_proj1_ppc0_bsp/ppc440_0:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/code:Z:
/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/include:Z:/Xps_proj3/SDK/SDK_Workspace/t
est_proj1_ppc0_bsp/ppc440_0/lib:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc
:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/apu_fpu_virtex5_v1_00_a:Z:/Xps
_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/apu_fpu_virtex5_v1_00_a/src:Z:/Xps_pr
oj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/bram_v3_01_a:Z:/Xps_proj3/SDK/SDK_Worksp
ace/test_proj1_ppc0_bsp/ppc440_0/libsrc/bram_v3_01_a/src:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1
_ppc0_bsp/ppc440_0/libsrc/common_v1_00_a:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440
_0/libsrc/common_v1_00_a/src:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/cp
u_ppc440_v2_01_a:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/cpu_ppc440_v2_
01_a/src:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/lldma_v2_00_a:Z:/Xps_p
roj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/lldma_v2_00_a/src:Z:/Xps_proj3/SDK/SDK_
Work-
space/test_proj1_ppc0_bsp/ppc440_0/libsrc/memcon_v2_00_a:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1
_ppc0_bsp/ppc440_0/libsrc/memcon_v2_00_a/src:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/pp
c440_0/libsrc/standalone_v3_07_a:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsr
c/standalone_v3_07_a/src:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/standa
lone_v3_07_a/src/profile:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/uartli
te_v2_00_a:Z:/Xps_proj3/SDK/SDK_Workspace/test_proj1_ppc0_bsp/ppc440_0/libsrc/uartlite_v2_00_a/src
:Z:/Xps_proj3/SDK/SDK_Workspace/Xps_proj3_hw_platform:Z:/Xps_proj3/SDK/SDK_Workspace/Xps_proj3_hw_
platform/cache:Z:/Xps_proj3/SDK/SDK_Workspace/Xps_proj3_hw_platform/settings:Z:/:$cdir:$cwd"
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote thread list.\n"
56^done
(gdb)
57^done,register-
names=["r0","r1","r2","r3","r4","r5","r6","r7","r8","r9","r10","r11","r12","r13","r14","r15","r16"
,"r17","r18","r19","r20","r21","r22","r23","r24","r25","r26","r27","r28","r29","r30","r31","f0","f
1","f2","f3","f4","f5","f6","f7","f8","f9","f10","f11","f12","f13","f14","f15","f16","f17","f18","
f19","f20","f21","f22","f23","f24","f25","f26","f27","f28","f29","f30","f31","pc","msr","cr","lr",
"ctr","xer","fpscr","","","","","","","","","","","","","","","","","pvr","","","","","","","","",
"","","","","","","","","","","","","sprg0","sprg1","sprg2","sprg3","srr0","srr1","tbl","tbu","","
","","icdbdr","esr","dear","ivpr","","tsr","tcr","dec","","","csrr0","csrr1","dbsr","dbcr0","iac1"
,"iac2","dac1","dac2","pir","rstcfg","mmucr","pid","ccr1","dbdr","ccr0","dbcr1","dvc1","dvc2","iac
3","iac4","dbcr2","sprg4","sprg5","sprg6","sprg7","decar","usprg0","ivor0","ivor1","ivor2","ivor3"
,"ivor4","ivor5","ivor6","ivor7","ivor8","ivor9","ivor10","ivor11","ivor12","ivor13","ivor14","ivo
r15","inv0","inv1","inv2","inv3","itv0","itv1","itv2","itv3","dnv0","dnv1","dnv2","dnv3","dtv0","d
tv1","dtv2","dtv3","dvlim","ivlim","dcdbtrl","dcdbtrh","icdbtrl","icdbtrh","mcsr","mcsrr0","mcsrr1
"]
(gdb)
58^done,depth="1"
(gdb)
59^done,bkpt={number="1",type="breakpoint",disp="del",enabled="y",addr="0x00000750",at="<exit+24>"
,times="0"}
(gdb)
```

```
60^done,bkpt={number="2",type="breakpoint",disp="del",enabled="y",addr="0x00000218",func="main",fi
le="../src/main1.c",fullname="Z:\\Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c"
,line="8",times="0"}
(gdb)
61^done,stack=[frame={level="0",addr="0x00000218",func="main",file="../src/main1.c",fullname="Z:\\
Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="8"}]
(gdb)
62^running
(gdb)
62*stopped,thread-
id="0",frame={addr="0x00000218",func="main",args=[],file="../src/main1.c",fullname="Z:\\Xps_proj3\
\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="8"}
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote thread list.\n"
63^done
(gdb)
64^done,depth="1"
(gdb)
65^done,stack=[frame={level="0",addr="0x00000218",func="main",file="../src/main1.c",fullname="Z:\\
Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="8"}]
(gdb)
66^done,changed-
regis-
ters=["0","1","2","3","4","5","6","7","8","9","12","13","14","15","16","17","18","19","20","21","2
2","23","24","25","26","27","28","29","30","31","33","64","65","66","67","69","87","108","109","11
0","111","113","114","119","121","122","124","129","131","132","133","134","135","136","137","143"
,"145","146","147","148","150","151","152","153","155","156","157","158","159","160","161","162","
163","164","165","166","167","168","169","170","171","172","175","180","181","182","183","190","19
1","192","193","195"]
(gdb)
&"info sharedlibrary\n"
~"No shared libraries loaded at this time.\n"
67^done
(gdb)
68^done,stack-args=[frame={level="0",args=[]}]
(gdb)
69^done,locals=[name="a",name="b",name="c"]
(gdb)
&"whatis a\n"
~"type = double\n"
70^done
(gdb)
&"whatis b\n"
~"type = double\n"
71^done
(gdb)
&"whatis c\n"
~"type = double\n"
72^done
(gdb)
73^done,name="var1",numchild="0",type="double"
```

```
(gdb)
74^done,value="2.1219815974473986e-314"
(gdb)
75^done,name="var2",numchild="0",type="double"
(gdb)
76^done,value="2.1219815974473986e-314"
(gdb)
77^done,name="var3",numchild="0",type="double"
(gdb)
78^done,value="-nan(0xf8fe800000000)"
(gdb)
79^done,value="-nan(0xf8fe800000000)"
(gdb)
80^done,value="0"
(gdb)
81^done,value="0"
(gdb)
82^running
(gdb)
82*stopped,reason="end-stepping-range",thread-
id="0",frame={addr="0x0000021c",func="main",args=[],file="../src/main1.c",fullname="Z:\\Xps_proj3\
\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="9"}
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote thread list.\n"
83^done
(gdb)
84^done,depth="1"
(gdb)
85^done,stack=[frame={level="0",addr="0x0000021c",func="main",file="../src/main1.c",fullname="Z:\\
Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="9"}]
(gdb)
86^done,changelist=[]
(gdb)
87^done,changelist=[]
(gdb)
88^done,changelist=[]
(gdb)
89^done,changed-
registers=["0","3","9","11","64","66","67","114","172","173","174","175","181","182","183"]
(gdb)
&"info sharedlibrary\n"
~"No shared libraries loaded at this time.\n"
90^done
(gdb)
91^done,stack-args=[frame={level="0",args=[]}]
(gdb)
92^done,locals=[name="a",name="b",name="c"]
(gdb)
93^running
(gdb)
```

```
93*stopped,reason="end-stepping-range",thread-
id="0",frame={addr="0x00000228",func="main",args=[],file="../src/main1.c",fullname="Z:\\Xps_proj3\
\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="10"}
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote thread list.\n"
94^done
(gdb)
95^done,depth="1"
(gdb)
96^done,stack=[frame={level="0",addr="0x00000228",func="main",file="../src/main1.c",fullname="Z:\\
Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="10"}]
(gdb)
97^done,changelist=[{name="var1",in_scope="true",type_changed="false"}]
(gdb)
98^done,changelist=[]
(gdb)
99^done,changelist=[]
(gdb)
100^done,changed-registers=["9","32","64","114","115","181","182","183"]
(gdb)
&"info sharedlibrary\n"
~"No shared libraries loaded at this time.\n"
101^done
(gdb)
102^done,stack-args=[frame={level="0",args=[]}]
(gdb)
103^done,locals=[name="a",name="b",name="c"]
(gdb)
104^done,value="1.2"
(gdb)
105^done,value="1.2"
(gdb)
106^running
(gdb)
106*stopped,reason="end-stepping-range",thread-
id="0",frame={addr="0x00000234",func="main",args=[],file="../src/main1.c",fullname="Z:\\Xps_proj3\
\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="11"}
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote thread list.\n"
107^done
(gdb)
108^done,depth="1"
(gdb)
109^done,stack=[frame={level="0",addr="0x00000234",func="main",file="../src/main1.c",fullname="Z:\
\Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="11"}]
(gdb)
110^done,changelist=[]
(gdb)
111^done,changelist=[{name="var2",in_scope="true",type_changed="false"}]
(gdb)
```

```
112^done,changelist=[]
(gdb)
113^done,changed-registers=["32","64","114","172","183"]
(gdb)
&"info sharedlibrary\n"
~"No shared libraries loaded at this time.\n"
114^done
(gdb)
115^done,stack-args=[frame={level="0",args=[]}]
(gdb)
116^done,locals=[name="a",name="b",name="c"]
(gdb)
117^done,value="2.2999999999999998"
(gdb)
118^done,value="2.2999999999999998"
(gdb)
119^running
(gdb)
119*stopped,reason="end-stepping-range",thread-
id="0",frame={addr="0x00000244",func="main",args=[],file="../src/main1.c",fullname="Z:\\Xps_proj3\
\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="12"}
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote thread list.\n"
120^done
(gdb)
121^done,depth="1"
(gdb)
122^done,stack=[frame={level="0",addr="0x00000244",func="main",file="../src/main1.c",fullname="Z:\
\Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="12"}]
(gdb)
123^done,changelist=[]
(gdb)
124^done,changelist=[]
(gdb)
125^done,changelist=[{name="var3",in_scope="true",type_changed="false"}]
(gdb)
126^done,changed-registers=["32","45","64","70","114","174"]
(gdb)
&"info sharedlibrary\n"
~"No shared libraries loaded at this time.\n"
127^done
(gdb)
128^done,stack-args=[frame={level="0",args=[]}]
(gdb)
129^done,locals=[name="a",name="b",name="c"]
(gdb)
130^done,value="2.7599999999999998"
(gdb)
131^done,value="2.7599999999999998"
(gdb)
132^running
```

```
(gdb)
132*stopped,reason="end-stepping-range",thread-
id="0",frame={addr="0x00000248",func="main",args=[],file="../src/main1.c",fullname="Z:\\Xps_proj3\
\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="14"}
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote thread list.\n"
133^done
(gdb)
134^done,depth="1"
(gdb)
135^done,stack=[frame={level="0",addr="0x00000248",func="main",file="../src/main1.c",fullname="Z:\
\Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="14"}]
(gdb)
136^done,changelist=[]
(gdb)
137^done,changelist=[]
(gdb)
138^done,changelist=[]
(gdb)
139^done,changed-registers=["0","9","10","64","66","67","114","115","172","173","174","180"]
(gdb)
&"info sharedlibrary\n"
~"No shared libraries loaded at this time.\n"
140^done
(gdb)
141^done,stack-args=[frame={level="0",args=[]}]
(gdb)
142^done,locals=[name="a",name="b",name="c"]
(gdb)
143^running
(gdb)
143*stopped,reason="end-stepping-range",thread-
id="0",frame={addr="0x0000024c",func="main",args=[],file="../src/main1.c",fullname="Z:\\Xps_proj3\
\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="15"}
(gdb)
&"info threads\n"
&"warning: RMT ERROR : failed to get remote thread list.\n"
144^done
(gdb)
145^done,depth="1"
(gdb)
146^done,stack=[frame={level="0",addr="0x0000024c",func="main",file="../src/main1.c",fullname="Z:\
\Xps_proj3\\SDK\\SDK_Workspace\\test_proj1_ppc0/Z/../src/main1.c",line="15"}]
(gdb)
147^done,changelist=[]
(gdb)
148^done,changelist=[]
(gdb)
149^done,changelist=[]
(gdb)
150^done,changed-registers=["0","64","114"]
```

```
(gdb)
&″info sharedlibrary\n″
~″No shared libraries loaded at this time.\n″
151^done
(gdb)
152^done,stack-args=[frame={level=″0″,args=[]}]
(gdb)
153^done,locals=[name=″a″,name=″b″,name=″c″]
(gdb)
&″kill\n″
154^done
(gdb)
155^exit
```

## References

[1] R. Avot-Chotin, H. Mehrez. „Hardware implementation of discrete stochastic arithmetic“. *Numerical Algorithms*, pp.21-33,2004.

[2] M.J. Schulte, E.E. Swartzlander Jr., „Hardware design and Arithmetic Algorithms for a Variable-Precision, Interval Arithmetic Coprocessor“. *Proceedings of the 12th symposium on computer arithmetic*, pp. 163-171, 1995.

[3] M.S. Cohen, T.E. Hull, and V.C. Hamarcher, „CADAC: A controlled-precision decimal arithmetic unit“. *IEEE Transactions on Computers,* vol . C-32, pp.370-377, 1983.d

[4] J. Vignes „Discrete stochastic arithmetic for validating results of numerical software”. *Numerical Algorithms* vol. 37, pp. 377–390, 2004.

[5] J.-M. Chesneaux, CADNA: „ An ADA tool for round-off errors analysis and for numerical debugging“. *Congres on ADA in Aerospace*, Barcelone, pp. 390–396. 1990.

[6] „CADNA for Fortran/C/C++ source codes“.URL: http://www-pequan.lip6.fr/cadna/

[7] Y. Baroud, „A Hardware Architecture for Numerical Instability Detection Based on Discrete Stochastic Alrithmetic“, University of Stuttgart, 2012.

[8] „IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008* , vol., no., pp.1-58, Aug. 29 2008 doi: 10.1109/IEEESTD.2008.4610935
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4610935&isnumber=4610934

[9] „EDK Concepts, Tools and Techniques“ URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/edk_ctt.pdf

[10] „Embedded System Tools Reference Manual“ URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/est_rm.pdf

[11] „GDB the GNU project debugger“ URL: http://www.gnu.org/software/gdb/

[12] J. Vignes, „Error analysis in computing“, *International Federation for Information Processing Congress*, Stockholm, August 1974, pp. 610–614.

[13] J. Vignes, „New methods for evaluating the validity of the results of mathematical computations“. *Math.Comput. Simulation 20 (1978)* pp. 227–249.

[14] J. Vignes, „ A stochastic arithmetic for reliable scientific Computation“,*Mathematics and Computers in Simulation, Vol. 35, Issue 3*, pp. 233–261,September 1993.

[15] W.Li, „Numerical Accuracy Analysis in Simulations on Hybrid High-Performance Computing Systems“, Technical report, Institute of Parallel and Distributed Systems, University of Stuttgart, 2010.

[16] R.E. Moore, „Interval Analysis“, *Prentice-Hall, Englewood Cliffs*, N.J., 1966.

[17] J.-M. Chesneaux, „Study of the Computing Accuracy by Using Probabilistic Approach“. *Contribution to computer arithmetic and Self Validating Numerical Methods*, pp. 19-30, 1990.

[18] „Debug overview, Xilinx software Development Kit Help Contents“ URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_3/SDK_Doc/index.html

[19] „Invoking GDB, GDB user manual“ URL: http://sourceware.org/gdb/current/onlinedocs/gdb/Invoking-GDB.html#Invoking-GDB

[20] „27 The GDB/MI interface“ URL: http://sourceware.org/gdb/onlinedocs/gdb/GDB_002fMI.html

[21] „Console (Windows), Windows Environment Development“ URL: http://msdn.microsoft.com/en-us/library/windows/desktop/ms682055(v=vs.85).aspx

[22] „Creation of a Console (Windows), Windows Environment Development“ URL: http://msdn.microsoft.com/en-us/library/windows/desktop/ms682528(v=vs.85).aspx

[23] „CreateProcess function, msdn“ URL: http://msdn.microsoft.com/en-us/library/ms682425%28v=VS.85%29.aspx

# Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

Kailai Wang, Stuttgart, 17 Dec, 2012