NORTH CAROLINA STATE UNIVERSITY

Raleigh, NC 27695, USA

GDB STUB DEVELOPMENT FOR H3927 BASED LEGO RCX SYSTEM

Fall 2002

CSC714 - 002 (Real-Time Computer Systems)
Instructor: Dr. Frank Mueller

Project Team Members:

Jaydeep Marathe	(j_marathe@hotmail.com)
Gautam Gopinadhan	(gvgopina@unity.ncsu.edu)
Palash Kasodhan	(pmkasodh@unity.ncsu.edu)

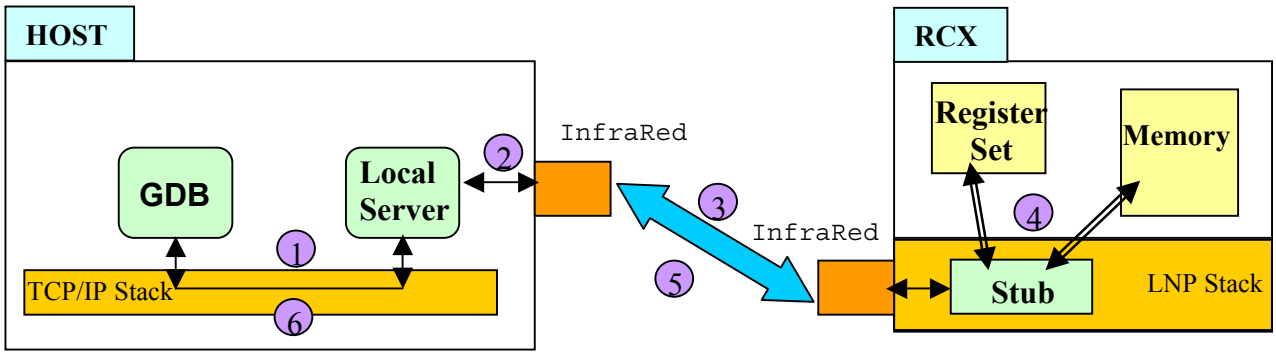
Document	*Project Report*	Ver. Rev.	*1.00a*
Authorized By	*Dr Frank Mueller, Assistant Professor, Dept of Computer Science, NCSU*	Signature/ Date	

TABLE OF CONTENTS

1 OBJECTIVE

This project aims at developing a gdb remote stub implementation for the LEGO Mindstorm RCX module with the h3297 processor. The current development/execution environment of the BrickOS system running on the RCX has no support for debugging; the project aims at rectifying this anomaly. The framework enables remote debugging of user and kernel level tasks, with the gdb program running on the host, communicating with the remote stub program over IR, through the gdb remote serial protocol.

2 FRAMEWORK OVERVIEW



1. GDB connects to LocalServer server through localhost TCP connection.
2. GDB sends command request to LocalServer, which relays it through IR to Stub on RCX. (in a *GDB Remote Serial Protocol* packet)
3. Stub reads and decodes incoming request from server application on host.
4. Stub executes requested operation (register read, breakpoint insertion, etc.)
5. Stub returns completion code (failure/success or requested data) to LocalServer.
6. Server on host returns completion code to GDB.

3 PREVIOUS WORK

3.1 Overview

Previous framework was designed by Les Smithson.
http://www.hare.demon.co.uk/lego/gdb.html).
The stub was derived from existing stub for the i386 architecture.

3.1.1 Working Components

- *Setup/ Breakdown* : Setup and breakdown of communication with host gdb under gdb serial protocol.(sending initial SIGTRAP signal packet to host gdb, acknowledging correctly received GDB packets, responding to GDB kill command)
- *LocalServer* : written in Python, to relay commands between gdb on localhost and stub on the RCX.
- *Query Commands* : Examining frames, registers, memory, reading and writing to registers and memory.

3.1.2 Non-working Components/Issues

- *Programmed Breakpoints :* The author was attempting to use the interrupt handler routine (ISR) for the spare IRQ2 interrupt as the software breakpoint instruction. This approach is fundamentally inadequate, since the instructions in the ROM ISR affect the condition code register (CCR), so user context is damaged and cannot be fully saved/restored.
- *Implicit Breakpoints :* Support for gdb-inserted (i.e. implicit) breakpoints was missing. Also, the approach of using IRQ2 interrupt ISR is unacceptable for implicit breakpoints for reasons discussed above.
- *Stepping :* Support for instruction level stepping was missing. This is non-trivial to implement, since the h3927 processor lacks trace bit support, so the stub has to be aware of the current instruction size and also needs to compute target address for transfer of control instructions (eg. branch, subroutine calls, return from subroutine, etc.)
- *Freezing Operating State :* Support for freezing other threads, inserting hooks for interrupts, etc. was missing.
- *Python Dependency :* There were issues with python interpreter version differences, due to which the shared library provided by the author could not be used with the python-encoded LocalServer, on a RedHat 8.0 machine. (though it worked correctly with a RedHat 7.2 machine with an older version of python (ver. 1.5))

4 CURRENT WORK

The current work is described for each significant task that was accomplished.

4.1 Task: Breakpoints

4.1.1 Objective

Devise mechanism for implementing software breakpoints. Mechanism should be consistent for both programmed breakpoints and gdb-inserted (i.e. implicit) breakpoints. It is essential that the mechanism save/restore complete user context, transparent to the target task, while minimizing the perturbation caused to the environment (in terms of execution overhead, memory usage).

4.1.2 Discussion of Constraints

- Target processor lacks any software interrupt instruction.
- Thus, we must transfer control to the breakpoint handler, either by direct subroutine calls (the JSR instruction), or by registering as interrupt handlers for interrupts. (both of which save the PC on the stack, so that we can eventually return from the breakpoint handler to the proper place in the target task).
- The condition code register (CCR), which contains the arithmetic and logic flags, is affected by almost all instruction, except for transfer of control instructions.
- The CCR must be saved and restored when jumping to and from, respectively, from the breakpoint handler.

Different solutions satisfying above constraints are discussed below.

4.1.3 Previous Solution (using IRQ2 software ISR handler)

- IRQ2 is not used on the RCX system. Set the breakpoint handler function as the software interrupt handler for IRQ2.
- Programmed breakpoints encoded as macro BREAKPOINT. The macro consists of : `pbreak=1; jsr @@0x0c; pbreak=0.`
- The pbreak variable allows breakpoint handler to distinguish between programmed and gdb-inserted breakpoints. The `jsr @@0x0c` is a subroutine call to the address stored in the double indirection of the ROM location `0x0c`. ROM location `0x0c` is the interrupt vector table (IVT) entry for IRQ2.
- The `jsr` call described above, calls the ROM interrupt service routine (ISR) handler, which in turn calls the RAM ISR handler.
- During initialisation, the RAM handler address had been set to the address of the breakpoint handler.
- **Problem :** The ROM ISR does not save the CCR (condition code register), so the CCR is trashed by the time control comes to the RAM handler. This makes this approach unacceptable for gdb-inserted breakpoints, since we must save/restore the complete processor state at any arbitrary breakpoint location.

4.1.4 With Direct JSR call to breakpoint handler (JSR _bpthandler)

- The breakpoint instruction is `JSR _bpthandler,` which translates to a 4-byte 'JSR to immediate 16-bit address' machine instruction.
- Programmed breakpoints are implemented as '`pbreak = 1; JSR _bpthandler; pbreak=0;`'.
- Modify gdb configuration file (*gdb/gdb/config/h8300/tmconfig.h*), and set the REMOTE_ BREAKPOINT instruction to the 4 byte machine opcode for 'JSR 0x0000'.
- When GDB host requests writes to memory, check if write requested corresponds to 'JSR 0x0000', and substitute it for the machine code bytes corresponding to '`JSR _bpthandler()`'.
- This substitution is required, since the address of `_bpthandler()` is not determined at compile time of host gdb, but only at run-time of target (if the stub is a part of the user space), or the compile time of the BrickOS kernel (if the stub is a part of the kernel).
- **Advantage :** No dependence on interrupt handlers, pure user-level code.
- **Problem :** The target processor has variable sized instruction – some are 2 bytes, while others are 4 bytes. Since the breakpoint instruction described above is 4 bytes, it will overwrite 2 instructions, instead of one instruction, if the instruction at breakpoint is a 2 – byte instruction. This makes handling of stepping difficult,and there were some special cases which could not be handled (eg. breakpoints on adjacent 2-byte instructions). Also, the substitution of the host-gdb write request of 'JSR 0x0000' into the machine code for 'JSR _bpthandler' must check whether the write is requested in the text section, before substituting. This is not foolproof, since there are no distinctions between text and data memory areas, and theoretically, the processor could execute from a data area.

4.1.5 With Register-indirect JSR call (JSR @Rx)

- Reserve one register from register set for debugging. This can be done by asking gdb not to use a particular register in code generation, using the *–ffixed-reg* flag. Care should be taken that this register is not r0,r1 or r2, since gcc passes parameters to functions in these registers (hand-written assembly routines in BrickOS assume this), r7 (since it's the stack pointer) or r6 (apparently its considered a scratch register by the hand-written assembly routines in BrickOS). We chose r5 as the debugging register.
- Compile the BrickOS kernel and user programs with gcc flag '*-ffixed-reg-r5*'.
- During stub initialisation, initialise r5 to contain address of *bpthandler()* as '`asm(" mov #_bpthandler,r5");`
- The breakpoint instruction is the 2 byte machine code of '`JSR @r5`' i.e. subroutine call to address held in register r5'.
- **Advantage:** Breakpoint instruction is 2-bytes, so it avoids complications caused due to 4-byte instruction size of previous approach.
- **Problem:** Processor already has low number of general purpose registers (r0-r6), reserving one register for debugging increases register pressure, leading to higher size of generated executables.

4.1.6 With BRA _here infinite loop and scheduler support (BRA –2)

- This approach suggested by Markus Noga.
- The breakpoint instruction is the 2-byte machine code for 'branch here', i.e. '`BRA –2`'

- When target hits breakpoint, it goes into infinite loop.
- Eventually, a timer interrupt is caused and control passes to BrickOS scheduler. Scheduler checks whether the interrupted task was in infinite breakpoint, and calls the breakpoint handler (`_bpthandler()`)
- **Advantage:** 2-byte instruction, so avoids complications due to variable instruction sizes. Also, no extra register is used, as in the previous approach.
- **Disadvantage**: potentially large time difference before control returns to breakpoint handler, since control transfer to breakpoint handler (`_bpthandler()`) happens only when the timer interrupt occurs; till then the target is stuck in an infinite loop. This probably will slow down response times significantly, especially when we are stepping the target at instruction level.

4.1.7 Selected Approach and Justification

- The last approach, using the 'BRA –2' infinite loop breakpoint instruction, was felt to be the best approach overall.
- However, we discovered the feasibility of this approach quite late, and we had gone ahead with the third approach, that of using 'JSR @r5' as the breakpoint instruction.
- The third approach has the slight disadvantage of using up one register for debugging, but a major advantage of having only 2-byte instruction size, thereby obviating complications caused due to variable size of instructions. Also, it is fairly transparent to the user (just need to add a flag (*-ffixed-reg-r5)* to the Makefile).
- The third approach is the best overall among the first three approaches and we have an implementation using it. We would also like to try out the fourth approach, which we feel is slightly superior technically.

4.1.8 A Note on Saving the CCR

- We must save the entire unmodified user context at breakpoint, i.e. the contents of the register set, pc and CCR at the start of the breakpoint handler, so that the user task can continue transparently, once we return from the breakpoint.
- Every 'move' instruction affects flags in the CCR. Also, the CCR can be saved only to a register, and not directly to memory.
- **Problem :** Since every move instruction affects CCR, we must save CCR as soon as possible. However, before we overwrite a register with the CCR in order to save it, we must save that register to memory using the 'move' instruction. This move also changes the flags in CCR.
- **Solution :** Transfer of control instructions do not affect the CCR. Also, move to memory instructions affect only the N (negative), Z (zero) and V (overflow) flags. Use the jump instructions to jump to unique points in the handler, depending on the unique combinations of the initial values of the NZV flags in the CCR. There are eight such unique locations. At each location, save R0 to memory, move CCR to R0. The initial move of R0 to memory affected the NZV flags, but due to our position in code due to the initial jumps described above, we know what the values of NZV were before they got trashed. So, we can patch these values back into the saved copy of the CCR, to get the original unmodified CCR value and save that to memory.

4.1.9 Overall Flow Sequence For GDB-inserted breakpoints

- Target program calls *debugInit()*,which has a programmed breakpoint at its end.
- When the programmed breakpoint is executed, control passes to the breakpoint handler.
- The breakpoint handler (*_bpthandler*) calls the *handle_exception* function of the stub which interfaces with the GDB host.
- User asks host GDB to insert breakpoint at particular address/line number, and requests continue (*'c'* command).
- Host GDB saves contents of target address using '*read memory*' request, handled by the stub, which is in the *handle_exception* function.
- Host GDB then overwrites contents at breakpoint address with the breakpoint instruction, using the '*write memory*' request handled by the stub, which is still in the *handle_exception* function.
- Host GDB sends '*continue*' request. On receiving the '*continue*' request, stub returns from *handle_exception* and target continues.
- When target reaches breakpoint address, the breakpoint instruction causes the *_bpthandler* to be called, using one of the approaches for breakpoint insertion discussed above.
- The *_bpthandler* saves user context, including CCR, and calls the *handle_exception* function of the stub.
- The *handle_exception* function sends signal SIGTRAP packet to host GDB.
- Host GDB restores original contents of breakpoint location, using the '*write* memory' request.
- Host GDB waits for user interaction.

4.2 Task : Stepping Functionality

4.2.1 Objective

The objective is to implement the stepping functionality, which allows the user to step through the program at machine instruction level granularity.

4.2.2 Discussion

- The target architecture lacks 'trace bit' functionality
- This means that the stub will have to be aware of the target instruction set architecture.
- The stub must know the size of the current instruction, so that it can decide where the next instruction is and insert a temporary breakpoint at that point.
- The stub must also decode the current instruction if it's a transfer of control instruction i.e. a branch (BCC), subroutine call (JSR, BSR), return (RTS,RTE) instruction, to determine if the transfer of control will take place. This is essential to determine the next instruction that will be executed, so that a temporary breakpoint can be inserted there.

4.2.3 Solution

- Stepping implementation was fairly straightforward.
- Isolated the instructions opcodes for 4-byte and 2-byte instruction sizes.
- Implemented reverse-engineering of transfer of control instructions to determine the next instruction that will be executed.

- Stepping has been tested and verified to work correctly.

4.2.4 Overall Flow Sequence For Stepping

- Intial condition :: remote target is in the handler for a breakpoint, and the remote stub is talking to host gdb, in the *handle_exception* stub function.
- User requests host gdb to step one machine instruction (using the '*stepi'* command).
- Host GDB requests stub to step one instruction using the '*step*' request.
- Stub calls the '*doSStep'* function to set up things for stepping.
- *doSStep* determines the size of the current instruction. Also determines whether the current instruction is a transfer of control instruction, and if so checks whether the transfer of control will happen or not, and the address of the branch target. Using this information, the function determines the address of the instruction that will be executed next, and inserts a breakpoint at that address. The original contents of this location are saved into temporary storage.
- The handler returns to the breakpoint address of the target and allows target to continue. The target executes current execution and encounters the breakpoint placed by *doSStep* at the very next instruction.
- In the breakpoint handler, the stub checks if it was stepping, and if so, calls the *undoSStep* function to restore the original contents of this instruction.
- Stub then contacts host GDB with *SIGTRAP* signal packet, which causes the host GDB to pass control to user.

4.3 Task : LocalServer Recoding in C

- The original LocalServer was coded in Python. It used an interface to the LNPD protocol through a shared library provided by the original author.
- We faced problems using this shared library on machines running RedHat 8.0 which has a later version of the python interpreter, while it ran without problems on machines with RedHat 7.2 , with python version 1.5 .
- We have recoded the LocalServer in C, using standard socket API, thus removing python dependency.

5 FUTURE WORK

There are some tasks that will enhance the debugger further, and increase its usefulness.

5.1 Minor Tasks

5.1.1 Freezing Environment State

Other threads (created using *execi* BrickOS call) should not be allowed to continue freely, when we are debugging a particular user task. This will entail minor modifications to the BrickOS scheduler to not schedule other tasks if so indicated.

5.2 Major Tasks

5.2.1 Thread Support

Threads on the BrickOS are created using the *execi* call. It would be useful to have a thread-cognizant gdb architecture. Currently, only single tasks can be debugged using GDB.

6 APPENDIX A : LIST OF TOOLS / PACKAGES

- On host side, need h8300 targeted binutils and gcc. We used gcc from egcs version 1.1.2,binutils tools version 2.9.5
- Need Lego Network Protocol Daemon (LNPD) and associated library liblnp for communicating with RCX using LNP protocol. The current versions available on the Internet work only with BrickOS (legOS) 2.4.
- LegOS 2.4 (due to dependency noted above).
- GDB 5.2.1 source package, since we need to change the breakpoint instruction for the h8300 target (as described in earlier sections).
- GDB 5.2.1 above compiled and built without problems on RedHat 7.2 and RedHat 8.0 systems with gcc 2.96 and gcc 3.2 respectively.

7 APPENDIX B : WEB RESOURCES

Here are some resources we found useful when doing the project.

- Previous work on remote debugging in RCX using gdb : http://www.hare.demon.co.uk/lego/gdb.html
- Description of Lego Network Protocol Daemon on Linux : http://legos.sourceforge.net/files/linux/LNPD/
- Description of GDB internals : http://www.cs.utah.edu/dept/old/texinfo/gdb/gdbint_toc.html
- Hitachi 8/300 Processor architecture and instruction set : http://www.informatik.hu-berlin.de/~mueller/rt/mindstorm/h3314.pdf
- GDB Online Book http://www.delorie.com/gnu/docs/gdb/gdb_toc.html
- Article on Remote Serial Protocol : http://www.redhat.com/devnet/articles/gdbtable1.htm

- Project Website : http://gopee.virtualave.net/rcx/

8 APPENDIX C : TASK DELEGATION

Task	Description	Responsibility	Status
Breakpoints	Provide support for programmed and gdb-inserted breakpoints. Build assembly breakpoint handler to save & restore target context.	Jaydeep	Completed
Stepping	Provide support for machine instruction granularity stepping. Involves mapping instruction set, and reverse engineering branch/subroutine instructions to decide next instruction to be executed.	Gautam and Palash	Completed
LocalServer Recoding	Recode LocalServer functionality in C from original Python code, to remove Python dependency	Gautam	Completed
Documentation	Compose Reports (1,2,3)	Palash and Jaydeep	Completed
Web page Maintenance	Maintain project web page	Gautam	Ongoing