X32-debug: A Remote Source-level Debugger for the X32 Soft Core

M. Dufour, S. Woutersen, A.J.C. van Gemund

Embedded Software Lab
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

November 2006

1 Introduction

A *debugger* allows programmers to investigate and influence running programs, making it a unique and important tool within the software development process. Advanced debuggers have two important properties in common. First, they allow the programmer to interact with their program at the familiar *source-code level*, so she does not have to deal with assembly language. Second, regarding (resource-constrained) embedded systems, advanced debuggers can typically be used *remotely*, e.g., through a serial link connected to a host PC.

The X32 [9] is a *soft core* developed at the TU Delft. To support software development on the X32, we have developed a *remote*, *source-level* debugger called "x32-debug". To allow for intuitive interaction, it comes with a simple *graphical user interface*. It was written with portability in mind: the operating system at the host PC can be easily changed, and its modular structure allows for easy replacement of any part (e.g., the X32 can be easily replaced by an alternative processor).

We have not ported an existing debugger, for three reasons. First, after investigating existing debuggers such as the GNU Debugger [2] and LDB [4], we concluded it would be less work to write a debugger from scratch using the Python programming language [6]. Second, a small, self-contained, program better fits an educational setting than a port of a huge, existing debugger, as it makes it easier to understand how the debugger works. Third, it shows how use modern tools, such as Python and PLY [1], to write a fairly advanced debugger in less than 1,200 lines of code (about 1% of the size of the GNU Debugger).

2 Overview

This section describes the global architecture of our debugger. It further describes some important aspects in more detail, such as the remote communication protocol, the implementation of user commands such as 'continue' and 'step', and finally the evaluation of source-level expressions.

2.1 Architecture

The following figure shows the global architecture of our debugger.

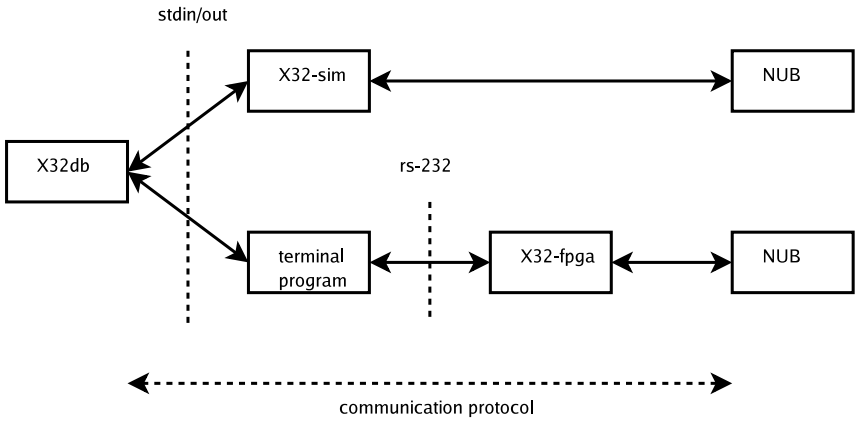

Figure 1: Global Architecture

The debugger is divided in two parts: one part running on the host PC, and one part running on the X32. The box on the left represents the part of the debugger that runs on the host PC. It provides the graphical user interface, reads program source code and debugging symbols (i.e., variable names for physical memory locations and source-level line numbers for physical program locations, as output by the linker), and processes user commands.

The two boxes on the right represents the tiny part of the debugger that runs 'remotely' on the X32, called a *NUB*. A NUB allows a debugger to remotely control a user program, by providing a small set of low-level primitives. Our NUB has five primitives: 'read/write memory block', 'set breakpoint', 'continue' and 'get processor state'. The part of the debugger that runs on the host PC uses sequences of these primitives to implement more high-level user commands such as 'continue' and 'step'. They also allow a primitive form of debugging with just a terminal program (e.g., Minicom [5]) at the host PC, should X32-debug not be available.

Communication between the host part and the NUB takes place via *standard in-* and *output* of the user program. When the NUB runs on the X32 *instruction simulator* (in the figure, x32-sim), this works out of the box; when the NUB runs on an actual X32 (e.g. on an FPGA board), standard in- and output are redirected via a *terminal program*, that connects to the X32 (in the figure, via a serial link, which is the typical means of communication with current FPGA boards). The NUB is activated when data arrives via standard input, i.e., when the host part sends a command, and uses standard output to send a reply (i.e., `printf`).

Communication between the host part and the NUB follows a well-defined *communication protocol*, as indicated in the lower part of the figure above. This communication protocol is further described in Section 2.2.

The host part is designed to be highly portable. It only uses platform-independent and open-source tools, such as Python, and any part of it can be easily replaced. Internally, it is divided into several small modules, according to the several tasks it must perform:

- Deal with processor-specific details, such as retrieving the processor state and unassembling
- Parse debug symbols and line information (in our case, as output by the LCC compiler [3])
- Communicate with the NUB, according to the used communication protocol (Section 2.2)
- Allow the user to evaluate arbitrary source-level expressions, such as `2+**b[0]*+*&a` (Section 2.4)
- Process user commands, using the above modules to perform the required tasks (Section 2.3)
- Deal with the graphical interface (highlighting current line and breakpoints, managing a command history, etc. [7, 8])

As noted, any of these modules can be easily replaced, i.e., to switch to a different processor, debug file format, communication protocol, user interface or programming language.

In all, we believe it is a testament to the power the the Python programming language and philosophy, that our complete debugger consists of less than 1,200 lines of code.

2.2 Communication Protocol

When the user program encounters a break point, it is suspended and the NUB is activated. The NUB then sends a ready signal to the host part, indicating it is ready to receive commands, and awaits further instructions. Before instructing the boot loader to start the program, X32-debug instructs the NUB to set a break point at "main".

Communication proceeds in a lock-step fashion. The host part sends a series of commands (corresponding to some high-level user command), each time waiting for a reply or acknowledgement from the NUB. In case of a 'continue' command, instead of sending a reply, the NUB returns control to the user program. This means the host part is blocked until another break point is triggered, causing a new ready signal to be sent. When the user program terminates, the boot loader sends a halt signal to the host part.

Our communication protocol requires the NUB to implement five commands. The following shows the format and meaning of these commands:

- *write memory block*: write a series of hexadecimally encoded bytes to a certain address. format: '`w address aabbcc..\n`'. example: '`w 1234 0000\n`' (clear 2 bytes at location 0x1234). reply: '`okay\n`'
- *read memory block*: read a sequence of bytes, starting at a certain address. format: '`r address number\n`'. example: '`r 1234 a\n`'. reply: '`aabbcc..\n`'.
- *processor state*: return the address of a memory location, where the state of the processor was stored, at the time the current break point was triggered. format: '`x\n`'. reply: '`address\n`'
- *set breakpoint*: set a break point on an instruction at a certain address; break points are removed by the NUB once they are triggered. format: '`b address\n`'. reply: '`okay\n`'
- *continue*: return control to the user program. format: '`c\n`'. reply: none (the host parts waits until it receives the ready signal, sent when another break point is triggered).

While the user program is running, it may accidentally send a ready or halt signal (e.g., using `printf`), confusing the host part. We avoid this for non-binary output, by prefixing these signals with the '`\a`' character.

We illustrate the communication protocol, by showing the exact messages sent between the host PC and the NUB, for a session involving the following example program (note that the exact sequence of messages depends on the used architecture):

```
1: int main() {
2:    printf(‘‘hello\n’’);
3:    printf(‘‘world.\n’’);
4:}
```

Suppose the user sets a break point on line 3, using the 'b' command (see Appendix A for an overview of all user commands) and continues twice, using the 'c' command. A sequence of messages similar to the following is now sent:

(*the user starts the debugger*)
< `ready` (the program has hit the break point at "main")

> `x` (the host parts requests the cpu state, including the current program counter)
< `1234` (address pointing to memory location with cpu state)
> `r 1234 20` (read the cpu state from memory)
< `aabbcc..` (the NUB sends the cpu state, and the host part deduces that we are at line 2)

(*line 2 is highlighted, and the user sets a break point at line 3*)
> `b 3456` (the host part knows the address of line 3 via the debug output of the linker)
< `okay` (the NUB has processed the command)

(*the user gives the 'continue' command*)
> `c` (the NUB returns control to the user program; the break point is not reset here; see Section 2.3)
< `hello` (the `printf` at line 2 is executed; the result is ignored by the host part)
< `ready` (the break point at line 3 is triggered)
> `x` (the host parts requests the cpu state, including the current program counter)
< `2345` (address pointing to memory location with cpu state)
> `r 2345 20` (read cpu state from memory)
< `aabbcc..` (the NUB sends the cpu state, and the host part deduces that we are at line 3)

(*line 3 is highlighted; the user again gives the 'continue' command*)
> `c` (the NUB returns control to the user program; see Section 2.3)
< `world` (the printf at line 3 is now executed; the result is ignored by the host part)
< `halt` (the host part now knows the user program has terminated)

(*no more line is highlighted, and the user is notified the program has terminated*)

2.3 User Commands

In this section, we describe our implementation of two user commands, that are more complicated than meets the eye. As noted, the 'continue' command is complicated by the need to reset the respective break point. The 'step' command is complicated by the need to step over individual machine instructions. We also describe how we deal with *interrupts* that can occur 'during' the execution of these commands.

Continue When the user continues from a break point, the respective instruction has to be executed, but at the same time, the break point must be maintained, because it may be triggered again. We solve the problem by setting a second break point on the instruction that is executed next. When the second break point is triggered, we reset the first break point, and continue as if nothing happened. The following figure illustrates the process:

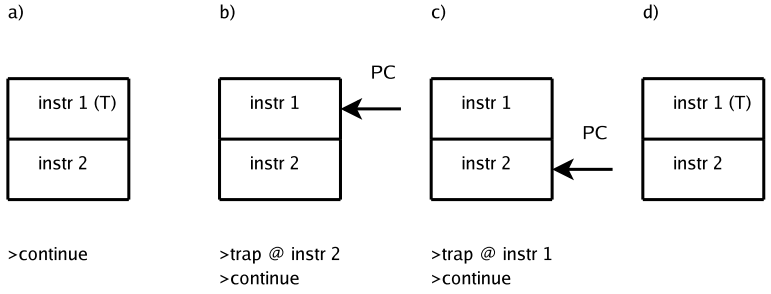

Figure 2: Continuing from a Break Point

Step a) shows the situation before the break point, indicated by '(T)', is triggered. Step b) shows the situation after it is triggered: the program counter now points to the first instruction, and the break point is removed. Step c) shows the situation after setting a break point on the next instruction, and continuing. Step d) shows the situation after resetting the original break point and continuing.

The situation is complicated further, because instructions may have different sizes, and the second instruction may not even by physically located after the first instruction (because the first instruction jumps somewhere else, e.g., being an X32 CALL or RET instruction). This means the host part needs to determine the address of the next instruction, based on either the size of the first instruction, or, in case of a jump, the current program state (which always determines the instruction to be executed next).

The following shows a sequence of messages, typically sent to continue from a break point:

(*the user gives the 'continue' command*)
> '`r 1234 6`' (read first instruction (assuming the maximum instruction size is 6))
< '`aabb..`' (first instruction, allowing the host part to figure out the next instruction, possibly after reading some mory from memory (e.g. in case of an indirect call instruction))
> '`b 123a`' (set break point on next instruction)
< '`okay`' (done)
> '`c`' (continue to next instruction)
< '`ready`' (now on next instruction)
> '`b 1234`' (reset original break point)
< '`okay`' (done)
> '`c`' (continue, resuming the user program)

(*the program now runs until another, or the same, break point is triggered*)

Step The 'step' command lets the user program continue, until it 'reaches' another (or the same) line of source code. Because of jump instructions (e.g., the X32 CALL and RET instructions), this may not be the subsequent line. The host part therefore has to figure out where to stop execution. Unfortunately, while the target of a jump instruction can often be figured out in advance, in general, it is not known until the last moment (e.g., consider an indirect call instruction).

We use the simplest possible solution to this problem, namely to single-step over individual machine instructions (using temporary break points). For each instruction, we determine the next instruction to be executed (using the same code as for the 'continue' command). If it is the first instruction of a line of source code, we stop execution here, allowing the user to step 'into' and 'out of' function calls. If the target of a jump instruction does not correspond to a line of source code (e.g., `printf`), we simply step 'over' the jump instruction.

Interrupts Interrupts can occur during the execution of user commands, interfering with their implementation. We assume the NUB disables interrupt handling from the moment it is triggered, until the moment it switches back to the user program. But this means interrupts can still occur at the first instruction executed for the 'continue' command (see above), and hence, for every instruction executed for the 'step' command. This is especially problematic for timer interrupts, that typically become pending during communication with the NUB, causing them to be handled immediately after switching back to the user program. This in turn causes programs control flow to jump in ways that are hard to understand for the user.

An intuitive approach to solve this problem is to simply disable all interrupts during execution of user commands such as 'continue' and 'step'. This does not work, however, for user programs that disable/enable interrupts themselves, without some complicated provisions. It is also not always clear when to re-enable interrupts, for example when stepping through a context-switch function:

it may take some time before we switch back to the same task, and all this time interrupts would be disabled. Indeed, by bluntly turning all interrupts off, we would interfere too much with the behaviour of some programs.

We have opted for a simpler solution, that interferes less with program behaviour. While it avoids the danger of already pending interrupts, it does not help against interrupts that become pending right after we switch back to the user program. For commands such as 'step' and 'continue', this means there is a very slight chance an interrupt can still interfere with their execution. The solution simply works by having the NUB clear all interrupts that have become pending after switching to the NUB, right before it switches back to the user program (the current implementation simply clears all pending interrupts). Intuitively, interrupts occurring during 'NUB time' are now ignored (avoiding the mentioned problem), while interrupts occurring during 'program execution' are never ignored (maintaining the correct program behaviour).

2.4 Expression Evaluation

Source-level debuggers can typically evaluate source-level expressions on behalf of the user, making it easier to investigate running programs. For example, an expression '`a+b`' shows the sum of two variables. The result of an expression depends on the state of the user program, e.g., the current program counter, and the memory state. This determines which variable '`a`' is meant (the same name may be used for a global and multiple local variables), its type (`char`, `int`, a pointer to a pointer to an `unsigned char`, etc.) and, finally, its current value.

There are two approaches to evaluate source-level expressions. One approach is to use an external compiler (typically the one used to compile the user program) to compile source-level expressions down to machine language, and somehow execute them within the current program state. We use a simpler approach, which consists of simply parsing and evaluating expressions ourself. A disadvantage of this approach is that it only works for a single high-level language (but we are mostly interested in C anyway).

To parse expressions, we use PLY [1], an implementation of the well-known parsing tools `lex` and `yacc` for Python. Lex allows one to parse the *lexical structure* of an expression, transforming it from a sequence of characters into a sequence of *tokens* (e.g., identifiers, numbers, keywords..), based on a certain *lexical grammar*. Yacc allows one to parse the *syntactical structure* of an expression, transforming it from a linear sequence of tokens into a tree-like structure called an *abstract syntax tree*, based on a certain *syntactical grammar*. An expression can be *evaluated* by performing a depth-first search over the nodes of the corresponding abstract syntax tree.

The nice thing about PLY is that we can specify the lexical grammar, the syntactical grammar, and the evaluation of abstract syntax trees in just a single, concise, pure Python file. We have written such a file, to allow for evaluation of expressions written in a substantial subset of the C language. All basic unary and binary operators are supported, and all types up to 4 bytes in length. Noteworthy omissions are that we currently do not support function calls inside expressions, casting operators and conditional expressions.

To evaluate an expression containing identifiers, we need to know which variables they point to, their types and current values. Having parsed the debug file as output by the linker, the main debugger program knows the types of all variables as well as their physical locations (which are relative, in case of local variables). It also knows how to communicate with the NUB to read their current values. Therefore, for each identifier, our evaluator simply asks the main debugger program about its type and current value.

References

[1] Beazley D., *Python Lex-Yacc*, `http://www.dabeaz.com/ply`

[2] Stallman R.M. et al., *The GNU Project Debugger*, `http://www.gnu.org/software/gdb`

[3] *LCC, a Retargetable Compiler for ANSI C*, http://www.cs.princeton.edu/software/lcc

[4] LDB, A Retargetable Debugger, http://www.eecs.harvard.edu/nr/ldb

[5] *Minicom Serial Communication Program*, http://alioth.debian.org/projects/minicom

[6] *The Python Programming Language*, http://python.org

[7] *The Qt Cross-Platform Application Development Toolkit*, http://www.trolltech.com/products/qt

[8] *The Scintilla Source Code Editing Component*, http://www.scintilla.org

[9] Woutersen S., *X32 Resource Site*, http://x32.ewi.tudelft.nl

A Usage

This appendix describes command-line options, user command syntax and available commands. Most information is copied from the 'help' and 'command' files distributed with the debugger. The command-line options are as follows:

```
Usage:  x32-debug [OPTION]...  FILE

-s --simulation        Use CPU simulator
-t --textmode          Use text mode interface
```

The command syntax is simply:

```
var = expr
command [expr1 [expr2 expr3 ..]]
```

Some general remarks about expressions:

- Expressions cannot contain whitespace characters.
- If omitted, expr1 evaluates to the address/line of the current instruction.
- An expression '#123' evaluates to the address of line 123.

The following finally describes individual user commands, in alphabetical order:

- `a`: print call stack

  Format: `a`

  Prints the current call stack

- `b`: set break point

  Format: `b [expr1 [expr2]]`

  Sets a break point at address expr1. Optionally, a condition expr2 can be given; the debugger evaluates the condition each time it passes the break point (which can be slow!), and only breaks if it is true.

  Examples:

  `b #` Set a break point at the current address
  `b 0x123` # Set a break point at address 0x123
  `b #44` # Set a break point at (the address of) line 44
  `b #44 i==j` # Set a break point, only to stop if "i==j" is true
  `b somefunc` # Set a break point at function "somefunc"

- `c`: continue

  Format: `c [expr1]`

  Continues program execution. If an optional address expr1 is given, a one-time break point is set here. This makes it easy to "jump" to a different address/line.

Examples:

```
c # Continue program execution
c #44 # Continue, breaking at (the address of) line 44
```

- `d`: dump memory contents

  Format: `d [expr1 [expr2]]`

  Dumps memory contents, from start address expr1 to optional end address expr2 (see 'o' command).

  Examples:

```
d 0x120 # Dump memory contents from 0x120
d 0x120 0x120+16 # Dump from 0x120 to 0x130
d &somevar &somevar+1 # Dump contents of variable
```

- `f`: run script file

  Format: `f expr1`

  Runs the lines in the file expr1 as though they are user commands.

  Example:

```
f blah # Run the lines in the file "blah" as commands
```

- `h`: evaluate expression (hex output)

  Format: `h [expr]*`

  Evaluates a list of expressions, separated by whitespace characters. Uses hexadecimal representation (see 'p' command).

  Examples:

```
h 1234 # Prints 1234 in hexadecimal
h a+1 # Evaluate single expression
h i[0] -j[0] i[0]+j[0] # Evaluate three expressions
```

- `i:` break point information

  Format: `i`

  Prints an overview of current break points.

- `l`: list source code

  Format: `l [expr1]`

  Lists source code from line expr1 (see 'o' command). Note: expr1 should evaluate to a line number, not an address.

Examples:

```
l # List source code starting at current line
l 44 # List source code starting at line 44
```

- `n`: skip to next line/instruction

  Format: `n`

  Skips to next line in source code, or to next instruction when in assembly mode (see 'z' command).

- `o`: set output length

  Format: `o expr1`

  Sets the output length for the 'd', 'l', 't' and 'u' commands.

  Example:

  `o 8` # Use output length of 8

- `p`: evaluate expression (decimal output)

  Format: `p [expr]*`

  Evaluates list of expressions, separated by whitespace characters (see 'h' command).

  Examples:

  ```
  p 0x1234 # Prints 1234 in decimal
  p a+1 # Evaluate single expression
  p i[0] -j[0] i[0]+j[0] # Evaluate three expressions
  ```

- `q`: quit debugger

  Format: `q`

  Quits the debugger, while resetting the program.

- `r`: restart application

  Restarts the application.

- `s`: skip to next line/instruction

  Format: `s`

  Skips to the next line in source code, or to the next instruction when in assembly mode (see 'z' command), while stepping in/out of function calls.

- `t`: show top of stack

Format: `t`

Shows the top of the stack (see 'o' command).

- `u`: unassemble

  Format: `u [expr1]`

  Unassemble (show assembly code) starting from address expr1 (see 'o' command).

  Examples:

  `u` # Unassemble from current address
  `u #44` # Unassemble from line 44
  `u somefunc` # Unassemble from start of function "somefunc"

- `w`: write to memory

  Format: `w expr1 [expr]*`

  Writes (byte) value(s) [expr]*, starting from address expr1.

  Examples:

  `w 0x123 4` # Writes 4 to address 0x123
  `w 0x123 1 2+2` # Writes 1 and 5, starting at address 0x123
  `w &i 1 2 3 -1` # Writes four bytes at the address of variable "i"

- `x`: show processor state

  Format: `x`

  Shows the current state of the processor.

- `z`: toggle assembly-level mode

  Format: `z`

  Toggles assembly-level mode. In assembly-level mode, the 'n' and 's' commands skip to the next instruction, instead of the next line.