Linux Development Kit For The Nios Embedded Processor

Application Notes VERSION 1.0

Issued: August 3, 2001

Debugging Applications

Starting the Debugger

The LDK only supports remote debugging; there is no self-hosted debugger available. Remote debugging consists of a debugger stub on the target, and a host debugger (with possibly a front-end such as DDD) on your workstation. The host communicates with the stub via a serial port running at 115,200 baud (8-bits, no parity, no flow control). Four shell-scripts have been provided to help set thing up for debugging:

GDBlinuxFlash : Host GDB for debugging a flash kernel
GDBlinux : Host GDB for debugging a RAM kernel
DDDlinuxFlash : Host DDD for debugging a flash kernel
DDDlinux : Host DDD for debugging a RAM kernel

A quick explanation of the GDB scripts follows. The DDD scripts differ only in the fact that they start the DDD front-end for GDB.

The first if-block in the script sets the script variables COM1 and COM2 to the serial port names depending on the environment. The Cygwin environment lays on top of Windows and defines the environment variable "niosgnu" whereas this symbol is absent in Linux environments. The if-block also defines an additional shell variable (GO_CMD) in the script for Flash-debugging. It is set to a null string in the Cygwin environment.

The next block creates a debugger startup command file that specifies the architecture (nios32), the file being debugged (linux), the port to use for remote debugging (COM2). The RAM script loads the kernel and sets a breakpoint at "main" whereas the Flash script will echo what to type to start the kernel. (The DDDlinuxFlash script will start automatically.) For either the GDB or DDD linuxFlash scripts to stop in the kernel startup procedure requires that the "#define"d symbol DEBUG_FLASH_KERNEL be non-zero (e.g. 1) in the file linux/arch/niosnommu/kernel/start.c. If it is changed, a kernel rebuild is naturally in order.

Next, the script loads the GDB stub into the target using the "nios-run -x" command. Normally, the flash loader will start linux after a reset: To prevent it from doing so and instead trapping to the GERMS monitor, you must depress and hold the SW4 button, press and release SW2 (reset), and a moment later (about 1/2 second), release SW4. The GERMS monitor will now be able to communicate with the host loader (nios-run): The GDB stub is loaded and started.

You can set breakpoints in the RAM-based kernel as soon as the debugger reports that the kernel is loaded. For flash-based systems, breakpoints can only be set _after_ the flash-loader loads the kernel since any breakpoints set _before_ the loader begins will be overwritten. If the kernel has been built with the DEBUG_FLASH_KERNEL symbol defined as 1 (see above), then the kernel will trap to the debugger shortly after the flash kernel has been loaded: Breakpoints may be set at this time or any time that the debugger has control (i.e. The kernel and applications are not running).

Finally, the script starts the host-side GDB debugger.

The default host-side debugger is nios-elf-gdb. If you desire, you can also run front-ends for gdb, such as ddd (http://www.gnu.org/software/ddd/). The DDDlinuxRAM and DDDlinuxFlash scripts are very similar to their GDBlinux* equivalents. Note, however, that the DDDlinux* scripts start the kernel running automatically, whereas this must be done manually with the GDBlinux* scripts. The command to type in the GDB command console will appear in the console window from which the GDBlinux* script was launched.

Preparing Applications for Debugging

There are two steps to preparing an application for debugging. The first is to patch the executable image with the utility "makeDebuggable" (found in the niosuserland/debug directory). This utility takes an optional argument of -y (the default) to make the following executable file-list debuggable, or of -n to "unpatch" the files. These patched executable files may be placed in the ROMFS in flash, however it is recommended that they instead be placed on an SMB server and mounted. Debugging typically requires several cycles of testing/rebuilding/reloading, etc. It takes a long time to burn a large ROMFS into flash.

The second step is to create a symbol file for the debugger to use. Add the following dependency to the "all" target in the makefile for your application: "app.abself", where "app" is the name of your executable. Please refer to niosuserland/hello/Makefile for an example.

Once the debugger is loaded and ready, run "nios-run -t" from a command window or run a terminal emulator (e.g. hyperterminal on Cygwin/Windows) set to 115,200 baud, no parity, 8-bits on com1 (Cygwin/Windows) or /dev/ttyS0 (linux). Start the kernel by typing "cont" (for a RAM-based kernel) or "jump * &na_flash_kernel" for a flash-based kernel in the debugger console window.

In the niosuserland/debug directory, there is a kernel utility named "debug" that the shell should have access to: By default, it is added to the /bin directory of the flash ROMFS, however it can live anywhere. If your executable and/or the "debug" utility are on an SMB server, mount the server.

The debug utility takes an argument of 0, 1, or 2 as follows:

"debug 0" - turn debugging off
"debug 1" - single-shot mode
"debug 2" - continuous mode

Single-shot mode will cause a trap to the debugger, but only once, when any executable patched for debugging is run. The debug mode is reset when the application traps to the debugger. To debug the next application, it is required that the debug utility be run again before the application is launched. Any applications _not_ patched for debugging will have no effect.

Continuous-mode will cause a trap to the debugger whenever any executable patched for debugging is run until the debugging mode is changed. Unpatched applications have no effect.

Now run your executable, and the host debugger will report "SIGTRAP". At this point, type "add-symbol-file <app> <text> <data> <bss>" at the debugger console prompt. <app> is the name of the .abself application file (e.g. hello.abself), and the <text>, <data>, and <bss> parameters are the addresses of the program sections. These values can be pasted from the terminal emulator. (The kernel loader prints them out whenever it causes a SIGTRAP.)

At the debugger console, type "step". This will move you from crt0 into __uClibc_main. You can now either directly set a breakpoint in your application and continue, or continue single-stepping into your application's "main()".

Tips

In some cases, your application may be out of scope to the debugger. The general advice is to "step" the debugger into __uClibc_main(), then "next" the debugger to the line where "main" is called: This will prevent you from stepping through things like the stdio initialization. Once you reach the point where "main" is called, "step" the debugger and you will arrive at "main()" in you application. At this point, the debugger is aware of your application's files.

Open the register and local variables windows when debugging. Examining a variable may display unexpected results because the variable may be cached in a register and not yet written to memory. The optimizer may also reorder operations. Sometimes the best course of action is a judicious sprinkling of printf statements in your code.

The host-side debugger sometimes has problems with paths to your source files. One way around this obstacle is to place symbolic links to your source files in the directory where the debugger is started, typically /opt/uClinux/linux.

It is a good idea to remove all breakpoints before your application terminates. Remember that your application is transient, and another application may well reuse it's memory. The debugger could then reinstate the old (now terminated) application's breakpoints in the new application's code space, or worse, restore the old application's instructions that were replaced with breakpoints.

Finally, check the README file enclosed with your kit for any late-breaking news of changes that have occurred since this note was published.