Debugging the kernel with the IDE

NOTE: This page is currently due for a re-visit to map against the latest kernel source/builds

---

The note describes the setup for using the IDE to debug the kernel on a remote target. The specific setup included a Linux host and PPC target although the same steps should apply for other (cross platform development) configurations as well.

In general, using the IDE to debug the kernel is no different that using the IDE to debug an application and it is suggested that you attempt to create a serial port (PDEBUG) launch configuration for a simple application first to ensure that you have connectivity and can actually launch and debug on your target. When you do have problems, this will provide you with the assurance that it is not connectivity issues. Don't under estimate the time involved in getting the correct serial port config and cables on your target and host, especially for new hardware.

Important:
It is assumed that you have the appropriate versions of all header files and libraries installed in your development environment. For this configuration, I had all of the required source checked out (including libc) and everything was built from within the IDE. Local modifications to common.mk files were made as required in order to point to the proper versions.

What you will need#

1. a target with 2 serial ports, one for the target console, the other for the debugger connection.
2. a host with at least one serial port for the debugger. If you have access to a second serial port you will have the added benefit of being able to monitor target console output
3. the appropriate serial cables

The steps are as follows (more details below)#

1. check out and build kdebug for the target
2. build a debuggable procnto
3. modify the target IFS build file to include the kdebug. Modify the startup command such that it will connect to the kdebug code and halt
4. build the target IFS
5. create the IDE launch configuration
6. connect the serial ports
7. reboot the target with the kdebug IFS
8. launch the debugger to attach to the kernel

Details

check out and build kdebug for the target#

the kdebug source is located in */product/services/kdebug*. From within the IDE, this source can be checked out as a QNX C/C++ project. Since I had the libc source checked out as well, I modified the project properties to include the location of the libc public directory *(from the properties page select QNX C/C++ Project. In the Compiler tab, Category drop down, select Extra Include Paths and add in the location of your libc source public directory)*. You may or may not need/want to do this. Just make sure you build against the the same libc version as built into your target IFS.

When building (even from within the IDE) I prefer to to create explict build targets as opposed to using the Build/Rebuild Project menu options. From the ./kdebug/gdb/ directory I right click on the ./ppc/ subdirectory (since I am building kdebug for a PPC based board) and select Make Build Target then create to add a new build target. Leave the target as 'all'. You may also want to create a 'clean' target.

```
make 'kdebug'
```

If successful, you should be left with a binary named 'gdb_kdebug-gdb'. This is the *'kdebug*' that must be added to the target IFS. I prefer to leave this file here and link to it from the target IFS build directory so that when I build a new IFS I always get the latest file. You can also copy it to the IFS build directory, just keep this in mind if you make any changes that may require *'kdebug*' to be rebuilt.

build a debuggable procnto#

Check out *./product/services/system/*. Only the following subdirectories are required. From the IDE, use a Standard C/C++ Project

- *./ker/*
- *./memmgr/*
- *./pathmgr/*
- *./proc/*
- *./procmgr/*
- *./public/*

A specific variant of procnto is built from within the *./proc/<target>/<variant>/* subdirectory. For my target for example, I can see by inspection of the default build file that it uses *'procnto-booke*'. To build the debug version I would build in the *./proc/ppc/be.booke.g/* subdirectory. In fact I will choose the *./proc/ppc/be.booke.inst.g/* subdirectory in order to get the instrumented debug version. Create a make target and build the procnto.

If successful, you should be left with a procnto binary'. This is the *'procnto*' that must be added to the target IFS. I prefer to leave this file here and link to it from the target IFS build directory so that when I build a new IFS I always get the latest file. You can also copy it to the IFS build directory, just keep this in mind if you make any changes that may require *'procnto*' to be rebuilt. Note that this is not the file which is loaded into GDB since there is no symbol information. A file which includes debugging information will be created when the IFS is created

modify the target IFS build file#

The next step is to build a target IFS that includes the "kdebug" and "procnto" binaries built in the previous steps
Set the kernel debug port to the desired configuration (-K option to startup)
add the kdebug binary (-K option will cause procnto to halt at startup)
point to the previously built procnto
Add the [+keeplinked] directive

For illustration, the PPC target build file modifications I made are shown below

```
 [image=0x100000]
 [virtual=ppcbe,raw] .bootstrap = {
    startup-85x0ads -v
    ###########################################################################
    ## PATH set here is the *safe* path for executables.
    ## LD_LIBRARY_PATH set here is the *safe* path for libraries.
    ##     i.e. These are the paths searched by setuid/setgid binaries.
    ##          (confstr(_CS_PATH...) and confstr(_CS_LIBPATH...))
    ###########################################################################
    PATH=:/proc/boot:/bin:/usr/bin LD_LIBRARY_PATH=:/proc/boot:/lib:/usr/lib:/lib/dll procnto-booke -v
```

```
 }
```

to

```
 [image=0x100000]
 [virtual=ppcbe,raw] .bootstrap = {
    startup-85x0ads -v -K8250.0xe0004600^0.115200
    ./gdb_kdebug-gdb -K
    ##########################################################################
    ## PATH set here is the *safe* path for executables.
    ## LD_LIBRARY_PATH set here is the *safe* path for libraries.
    ##     i.e. These are the paths searched by setuid/setgid binaries.
    ##          (confstr(_CS_PATH...) and confstr(_CS_LIBPATH...))
    ##########################################################################
 [+keeplinked]
    PATH=:/proc/boot:/bin:/usr/bin LD_LIBRARY_PATH=:/proc/boot:/lib:/usr/lib:/lib/dll   /home/mkisel/workspace2/nto/HEAD
 }
```

Careful: the /<path to procnto>/procnto-xxx_g must be on the same line as the PATH statement or a .sym file will not be produced

build the target IFS#

mkifs the build file to produce the IFS as you normally would.

The [+keeplinked] directive will cause a *'procnto-booke-instr_g.sym*' file to be produced. This is the file that is used by the debugger when it is launched. I found it useful to create a link from my ./services/system/proc/<cpu>/<procnto variant>/ directory to the .sym file so that it is easily located by the launch configuration tool.

create IDE launch configuration#

- Open Launch Configurations dialog (Run->Open Debug Dialog...)
- From the Debug screen, right click on *'C/C++ Local Application* and create a New launch configuration.
- In the Main tab, choose the project which contains the procnto built above. If you created a link to the .sym file above, the *'procnto-xxx.sym*' file should appear as the debug executable ... select it. If you do not yet have a target, create one that uses the Local QConn connector.
- In the Debugger tab, choose "gdbserver Debugger" from the Debugger drop down menu list. In the Debugger Options Main tab, change the GDB Debugger to be *'nto<cpu>-gdb*' from *'gdb*' and fill in the serial port configuration in the "Connections" tab. The serial port is /dev/ttyS0 for the Linux hosted IDE. I believe it is the same for Windows hosted IDE. If you have more than one serial port on your host, the serial port selected should be the one that is connected to the second (non console) target serial port.
- In the Source tab, select any source code locations not already established

launch the debugger to attach to the kernel#

Configure the target to boot the IFS built (see build the target IFS above). You should see a message output on the target console that looks similar to this ...

```
System page at phys:0000d000 user:0000d000 kern:0000d000
Starting next program at v00112158
KDEBUG at 0x146780, (TRAP) S/C/F=5/1/3
```

procnto is waiting for a connection from the debugger. Launch the debug session. At this point you should have a connection to the target procnto. In the debug perspective, execution should be halted at the symbol _start(). No source file will be found. Browse to the file ./services/system/ker/_main.c and set a breakpoint on _main(). At this point you can single step or resume execution. You should hit the _main() breakpoint.

Have fun!

Other Stuff#

- I have noticed that the debugger loses its connection with the target more often than normal QConn (IP) use. Have not yet determined the cause of this. Fortunately in most cases all that is required is to relaunch the debugger and you can carry on where you left off.
- I have not done this yet, but if you can I would suggest setting up flow control (preferably hardware) on the debug serial port connection. There is a lot of messaging being exchanged and I am not sure how robust the GDB/KDebug protocol is. If you can it wouldn't hurt and may allow more reliable behaviour at the highest baud rates.
- make sure that you do not have any startup config that will overwrite the debug serial port configuration (devc-xxx)
- make sure that you remove any expressions and/or breakpoints from previous debug sessions as this seems (anecdotally anyway) to cause more instability
- in general single stepping and reading variables etc takes longer when using the serial connection. Be patient, the more clicks the more likely you will cause the debugger to disconnect. It is generally better to use Run To Line (or breakpoint), examine some data and then Run To Line (or breakpoint) rather than single stepping