Yocto Project Summit 2021

Leveraging the Yocto Project to debug an embedded Linux system

Sergio Prado

Toradex

$ WHOAMI

- Designing and developing embedded software for 25+ years.
- Software Team Lead at Toradex (https://www.toradex.com/).
- Consultant/trainer at Embedded Labworks (e-labworks.com/en).
- Open source software contributor, including Buildroot, Yocto Project and the Linux kernel.
- Sometimes write technical stuff at https://embeddedbits.org/.
- Social networks:
  Twitter: @sergioprado
  Linkedin: https://linkedin.com/in/sprado

AGENDA

1. Quick introduction to debugging
2. Preparing the host and the target for debugging
3. Hands-on!

THIS TALK IS NOT ABOUT...

- ✗ Debugging BitBake metadata (recipes, classes, etc).
- ✗ Debugging build problems.
- ✗ How debugging tools work.

This talk will cover tips and tricks on debugging kernel and userspace applications on Linux systems generated with OpenEmbedded and the Yocto Project.

DEBUGGING TOOL AND TECHNIQUES

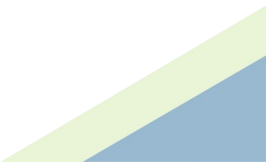
- **Logging and memory dump analysis**: dmesg, kernel oops/panic, addr2line, core dump, etc.
- **Interactive debugging**: GDB, KGDB.
- **Tracing and profiling**: ftrace, systemtap, perf, LTT-NG, gprof, strace, ltrace, etc.
- **Debugging frameworks**: kmemleak, valgrind, mtrace, etc.

APPROACHES TO DEBUGGING

- There are a few approaches to debugging an embedded Linux device:
  - Directly on the target (may not be possible, depending on available resources).
  - Remote debugging via some connection to the target (network, serial port, etc).
  - Post-mortem analysis (here we collect debug information in the target but do the analysis in the host).

WHAT WE (MIGHT) NEED TO DEBUG

- **Source code** of the binary (and libraries) to be debugged.
- Binary (and libraries) compiled with **debugging symbols**.
- Binary (and libraries) compiled with **optimizations disabled**.
- Binary (and libraries) compiled with **security mitigations disabled**.
- Many host and target **tools**!



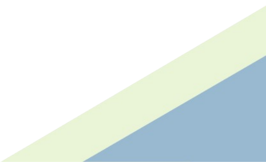
SOURCE CODE

- The source code of the binary (and libraries) is not shipped in the final image, but may be needed by a debugging tool like GDB.
- The source code can be included in the image with the `-src` package.
- Including all source packages:

```
IMAGE_FEATURES_append = " src-pkgs"
```

- Including the source package of a specific application:

```
IMAGE_INSTALL_append = " busybox-src"
```



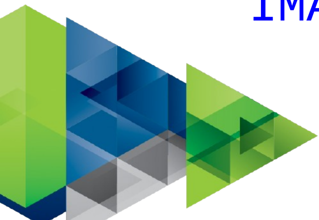
DEBUGGING SYMBOLS (1)

- Debug symbols are special entries in the symbol table of an object file that make it possible for tools like GDB and addr2line to gain access to information from the source code of the binary (and ultimately convert addresses into file names and line numbers).
- On Yocto, debugging symbols are removed by default from binaries and split into a separate package (`-dbg`).
- Including all debug packages:

```
IMAGE_FEATURES_append = " dbg-pkgs"
```

- Including the debug package of a specific application:

```
IMAGE_INSTALL_append = " busybox-dbg"
```

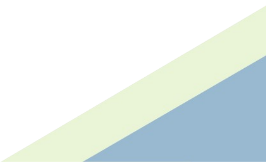


DEBUGGING SYMBOLS (2)

- Another option is to just disable stripping binaries (usually not necessary if you have the debug packages installed):

```
INHIBIT_SYSROOT_STRIP_pn-busybox = "1"
```

- If you are doing remote debugging, symbol resolution will be done in the host, and probably you don’t need to care about installing debugging packages in the target.



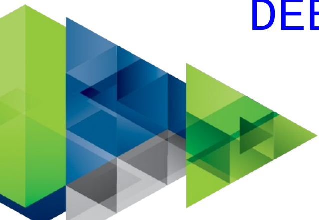
OPTIMIZATION (1)

- When building an application, the compiler may generate an optimized binary that does not match the source code, impacting the debug experience.
- On Yocto, the variable SELECTED_OPTIMIZATION specifies the optimization flags passed to the C compiler when building for the target.
- SELECTED_OPTIMIZATION takes by default the value of FULL_OPTIMIZATION:

```
FULL_OPTIMIZATION = "-O2 -pipe ${DEBUG_FLAGS}"
```

- If DEBUG_BUILD="1", then the variable SELECTED_OPTIMIZATION will take the value of DEBUG_OPTIMIZATION:

```
DEBUG_OPTIMIZATION = "-Og ${DEBUG_FLAGS} -pipe"
```



OPTIMIZATION (2)

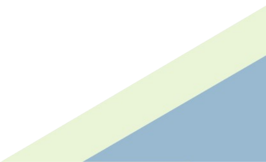
- Compiling all applications without optimization:

```
DEBUG_BUILD = "1"
```

- Compiling a specific application without optimization:

```
DEBUG_BUILD_pn-busybox = "1"
```

- Be aware that disabling optimizations may influence the system's behavior (it may even hide bugs!), so do it only when necessary.



SECURITY MITIGATIONS

- If your distro is building applications with security flags enabled, the debugging experience may be impacted.
- This is true for Poky-based distros (currently includes `security_flags.inc`), which enables security-related compiler flags by default.

```
require conf/distro/include/security_flags.inc
```

- In particular, if PIE (Position Independent Executable) is enabled, we are not able to easily resolve symbols, and you may want to disable it for a specific application during a debugging session:

```
SECURITY_CFLAGS_pn-busybox = "${SECURITY_NOPIE_CFLAGS}"
```

TARGET DEBUGGING TOOLS (1)

- Adding GDB to the rootfs for native debugging:

```
IMAGE_INSTALL_append = " gdb"
```

- Adding GDB server to the rootfs for remote debugging:

```
IMAGE_INSTALL_append = " gdbserver"
```

- The same can be achieved by adding `tools-debug` to `IMAGE_FEATURES` (it will add `gdb`, `gdbserver`, `mtrace` and `strace`):

```
IMAGE_FEATURES_append = " tools-debug"
```

TARGET DEBUGGING TOOLS (2)

- Adding Valgrind to the rootfs for memory analysis:

```
IMAGE_INSTALL_append = " valgrind"
```

- Adding profiling tools (perf, blktrace, powertop, systemtap, lttng, valgrind, etc):

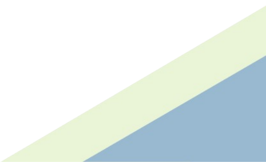
```
IMAGE_FEATURES_append = " tools-profile"
```

- Enable Eclipse debugging (gdbserver, tcf-agent, openssh-sftp-server):

```
IMAGE_FEATURES_append = " eclipse-debug"
```

- For easy connection to the target, `debug-tweaks` is useful:

```
EXTRA_IMAGE_FEATURES = "debug-tweaks"
```



HOST DEBUGGING TOOLS (1)

- To do remote debugging in the host, you need a **sysroot** (including binaries with debugging symbols) and **debugging tools**.

- One can generate a stripped-down version of the filesystem for debugging using the following variables:

```
IMAGE_GEN_DEBUGFS = "1"
IMAGE_FSTYPES_DEBUGFS = "tar.bz2"
```

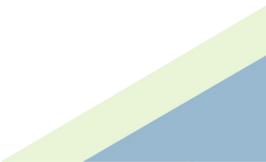
HOST DEBUGGING TOOLS (2)

- Then a minimal toolchain with only the tools can be generated with:

```
$ bitbake meta-toolchain
```

- Another approach is to generate a complete SDK, that will contain the sysroot with debugging symbols and all needed tools:

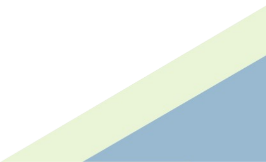
```
$ bitbake -c populate_sdk <your-image>
```



KERNEL SPACE DEBUGGING (1)

- Kernel space debugging is not that different from userspace debugging, but may require specific kernel configuration options enabled and additional tools.

- In particular, changing the kernel configuration to enable debug options may be required.



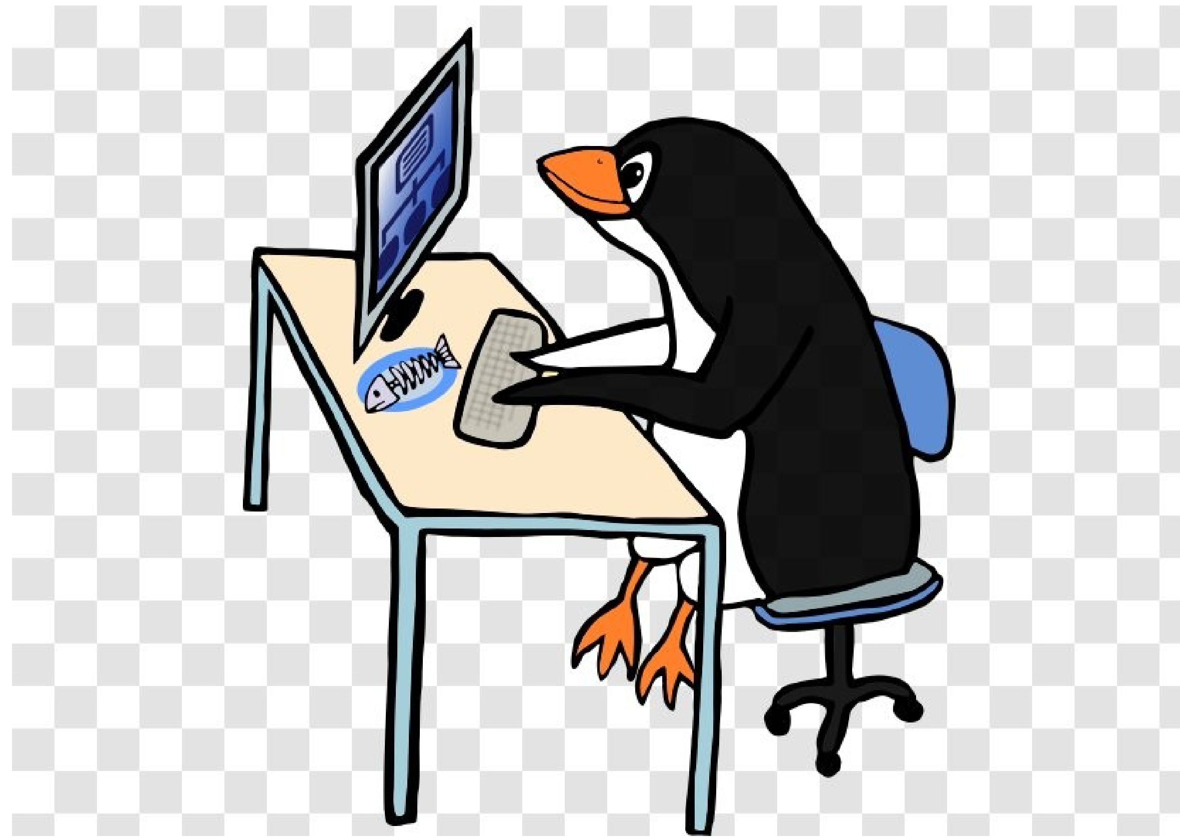
KERNEL SPACE DEBUGGING (2)

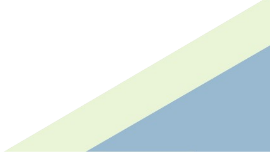
- You may need to create a specific `defconfig` for your kernel or a kernel config fragment (in case you are leveraging `kernel-yocto.bbclass`):

```
$ bitbake virtual/kernel -c kernel_configme -f
$ bitbake virtual/kernel -c menuconfig
$ bitbake virtual/kernel -c diffconfig
```

- For proper kernel debugging, you will probably want the kernel ELF image (`vmlinux`) compiled with debugging symbols.
  - For that, we need to enable `CONFIG_DEBUG_INFO`.
  - The image with debugging symbols will be in the kernel build directory (`B`).

HANDS-ON!





Q&A

Sergio Prado
sergio@embeddedbits.org

https://twitter.com/sergioprado
https://www.linkedin.com/in/sprado

Thank you!