External Pre-built Binary Toolchains in Yocto Project

Denys Dmytrieenko
LCPD, Arago Project
Texas Instruments
Definitions 1/2

• What is “External Pre-built Binary Toolchain”?
  – A cross-compilation toolchain (compiler, linker, assembler, libc, etc.) that is acquired from a third-party in the form of binary executables and libraries and is not built by the Yocto Project as part of the normal target build process.

• Why so many qualifiers?
  – External (vs. Internal to Yocto Project)
  – Pre-built (vs. Built as part of the target build process)
  – Binary (vs. Sources that are built)
Definitions 2/2

- What is LCPD?

- Linux Core Product Development
3rd Party Toolchains

Popular:

- CodeSourcery Sourcery G++ Lite
  - (now Mentor Graphics Sourcery CodeBench Lite)

- Linaro Toolchain Binaries
  - [https://launchpad.net/linaro-toolchain-binaries](https://launchpad.net/linaro-toolchain-binaries)

Less known:

- Angstrom toolchain

- Arago toolchain
Existing Support

- **TCMODE variable**
  - Sets `PREFERRED_PROVIDER` for toolchain components:
    ```
    virtual/${TARGET_PREFIX}gcc
    virtual/${TARGET_PREFIX}g++
    virtual/${TARGET_PREFIX}gcc-initial
    virtual/${TARGET_PREFIX}gcc-intermediate
    virtual/${TARGET_PREFIX}binutils
    virtual/${TARGET_PREFIX}compilerlibs
    virtual/${TARGET_PREFIX}libc-for-gcc
    virtual/${TARGET_PREFIX}libc-initial
    virtual/libc, virtual/libintl, virtual/libiconv
    linux-libc-headers, gdb, gdbserver...
    ```

- **recipes-core,eglibc,eglibc-package.inc**
  - Handles most of toolchain packaging work

- **Still requires recipe for scrapping/sysroots/packaging**
  - `external<-name>-toolchain.bb`

- **History in OE-Classic?**
Using CodeSourcery

- Part of OpenEmbedded-Core
  - Poky -> Arago (CSL_VER_*) -> OE -> OE-Core
- \texttt{TCMODE = "external-sourcery"}
  - Or the old name “external-csl”
- \texttt{EXTERNAL_TOOLCHAIN = "/path/to/csl"}
- Version agnostic – \texttt{CSL_VER_*}
- Extras:
  - Supports ARM, MIPS, PowerPC
  - Supports multilib (e.g. armv4/5/6, but not v7)

» Richard -> Denys -> Tom -> Chris
Using Linaro

- Part of meta-linaro layer:
  - git://git.linaro.org/openembedded/meta-linaro.git
- TCMODE = “external-linaro”
- EXTERNAL_TOOLCHAIN = “/path/to/linaro”
- Latest binaries are ARM hardfp:
  - ELT_TARGET_SYS ?= “arm-linux-gnueabihf”
  - Otherwise version agnostic – ELT_VER_*

» Ken Werner -> Marcin (shout-out to Khem)
Using Own, e.g. Arago

- Part of meta-arago (extras) layer
  - git://arago-project.org/git/meta-arago.git
- TCMODE = "external-arago"
  - Version agnostic - ARG_VER_* , ARG_LIC_*
- Arago distro in meta-arago finds toolchain in
  PATH and sets EXTERNAL_TOOLCHAIN and
  TOOLCHAIN_PATH

recipes-core/meta/external-arago-toolchain.bb
Adding Own, e.g. Arago 1/2

require recipes-core/eglibc/eglibc-package.inc
INHIBIT_DEFAULT_DEPS = "1"

PROVIDES = "\n  virtual/${TARGET_PREFIX}gcc \n  virtual/${TARGET_PREFIX}g++ \n  virtual/${TARGET_PREFIX}gcc-initial \n  virtual/${TARGET_PREFIX}gcc-intermediate \n  virtual/${TARGET_PREFIX}binutils \n  virtual/${TARGET_PREFIX}libc-for-gcc \n  virtual/${TARGET_PREFIX}compilerlibs \n  virtual/libc virtual/libintl virtual/libiconv glibc-thread-db \n  libgcc linux-libc-headers linux-libc-headers-dev gdbserver"

PACKAGES = "\n  libgcc libgcc-dev libstdc++ libstdc++-dev \n  linux-libc-headers-dev gdbserver glibc ldd glibc-utils"
Adding Own, e.g. Arago 2/2

FILES_glibc = "\n  ${sysconfdir} ${libexecdir}/* /lib/libc* /lib/libm* /lib/ld* \n  /lib/libpthread* /lib/libresolv* /lib/librt* /lib/libutil* \n  /lib/libnsl* /lib/libnss_files* /lib/libnss_compat* /lib/libdl* \n  /lib/libnss_dns* /lib/libanl* /lib/libBrokenLocale* \n  /sbin/ldconfig"

DESCRIPTION_* = "..."
PKG_${PN}_* = "eglibc*"...
PKGV_* = "${ARG_VER_*}"...
LICENSE_* = "${ARG_LIC_*}"...

do_install() {
  install -d ${D}${bindir}
  install -d ${D}${libdir}
  install -d ${D}${includedir}
  cp -a ${EXTERNAL_TOOLCHAIN}/${TARGET_SYS}${libdir}/* \n    ${D}${libdir}
  cp -a ${EXTERNAL_TOOLCHAIN}/${TARGET_SYS}${includedir}/* \n    ${D}${includedir}
  ...
}
Issues/Limitations

• TCLIBC
  – LIBC_DEPENDENCIES
  – external-<name>-toolchain.bb generates eglibc* packages, but “PROVIDES” glibc.

• SDK
  – By default, only libc libraries and headers are installed in sysroots and packaged for target
  – The actual host cross-sdk/cross-canadian toolchain binaries (gcc, binutils, gdb) are not packaged
TCLIBC

- Either modify all external toolchain recipes with
  `PROVIDES = "eglibc*"
- Or add and set supplemental TCLIBC, e.g.
  `tclibc-external-arago-toolchain.inc` that tweaks libc vars,
  like `LIBC_DEPENDENCIES`:

```
LIBC_DEPENDENCIES = "\n    libsegfault \n    glibc \n    glibc-dbg \n    glibc-dev \n    glibc-utils \n    glibc-thread-db \n    ${@get_libc_locales_dependencies(d)}"
```
Packaging SDK, Configuration

- Since libc is already packaged, just pull in required -dev
- Configure preferred providers for those cross-sdk/cross-canadian binaries, e.g. in `tcmode-external-<name>`:

  ```
PREFERRED_PROVIDER_gcc-cross-canadian-`{TRANSLATED_TARGET_ARCH}` = "external-<name>-sdk-toolchain"
PREFERRED_PROVIDER_gdb-cross-canadian-`{TRANSLATED_TARGET_ARCH}` = "external-<name>-sdk-toolchain"
PREFERRED_PROVIDER_binutils-cross-canadian-`{TRANSLATED_TARGET_ARCH}` = "external-<name>-sdk-toolchain"
```
external-<name>-sdk-toolchain.bb:

inherit cross-canadian

PROVIDES = "gcc-cross-canadian-\${TRANSLATED_TARGET_ARCH} \n    gdb-cross-canadian-\${TRANSLATED_TARGET_ARCH} \n    binutils-cross-canadian-\${TRANSLATED_TARGET_ARCH}" 

PACKAGES = "gcc-cross-canadian-\${TRANSLATED_TARGET_ARCH} \n    gdb-cross-canadian-\${TRANSLATED_TARGET_ARCH} \n    binutils-cross-canadian-\${TRANSLATED_TARGET_ARCH}" 

FILES_gcc-cross-canadian-\${TRANSLATED_TARGET_ARCH} = "\n    ${prefix}/${TARGET_SYS}/bin/cpp \n    ${prefix}/${TARGET_SYS}/bin/cc \n    ${prefix}/${TARGET_SYS}/bin/g++ \n    ${prefix}/${TARGET_SYS}/bin/gcc \n    ${prefix}/${TARGET_SYS}/lib/libstdc++.* \n    ${prefix}/${TARGET_SYS}/lib/libgcc_s.* \n    ${gcclibdir}/${TARGET_SYS}/${ARG_VER_GCC}/* \n    ${bindir}/${TARGET_PREFIX}gcov \n    ${bindir}/${TARGET_PREFIX}gcc \n    ${bindir}/${TARGET_PREFIX}g++ \n    ${bindir}/${TARGET_PREFIX}cpp \n    ${libexecdir}/*"
Packaging SDK, Recipe 2/3

FILES_gdb-cross-canadian-\{TRANSLATED_TARGET_ARCH\} = "\$
${bindir}/${TARGET_PREFIX}gdb \$
${bindir}/${TARGET_PREFIX}gdbtui \$
${datadir}/gdb/*"

FILES_binutils-cross-canadian-\{TRANSLATED_TARGET_ARCH\} = "\$
${prefix}/${TARGET_SYS}/bin/ld \$
${prefix}/${TARGET_SYS}/bin/objcopy \$
${prefix}/${TARGET_SYS}/bin/readelf \$
${prefix}/${TARGET_SYS}/bin/nm \$
${prefix}/${TARGET_SYS}/bin/as \$
${bindir}/${TARGET_PREFIX}ld \$
${bindir}/${TARGET_PREFIX}objcopy \$
${bindir}/${TARGET_PREFIX}readelf \$
${bindir}/${TARGET_PREFIX}nm \$
${bindir}/${TARGET_PREFIX}as \$
${includedir}/*.h ${libdir}/ldscripts/* ${libdir}/libiberty.a"
Packaging SDK, Recipe 3/3

PKGV_* = "${ARG_VER_*}"...
LICENSE_* = "${ARG_LIC_*}"...

do_install() {
    install -d ${D}${prefix}/${TARGET_SYS}/bin
    install -d ${D}${prefix}/${TARGET_SYS}/lib
    install -d ${D}${bindir}
    install -d ${D}${libdir}
    install -d ${D}${includedir}
    install -d ${D}${libexecdir}
    install -d ${D}${gcclibdir}/${TARGET_SYS}/${ARG_VER_GCC}/include
    cp -a ${TOOLCHAIN_PATH}/${TARGET_SYS}/bin/{cpp,cc,g++,gcc} \ ${D}${prefix}/${TARGET_SYS}/bin
    cp -a ${TOOLCHAIN_PATH}/${TARGET_SYS}/lib/{libstdc*,libgcc_s*} \ ${D}${prefix}/${TARGET_SYS}/lib
    ...
}
Rolling Own Binary Toolchain

• Use or extend one of `meta-toolchain*.bb` recipes
• Make sure to use “internal” toolchain settings to build one from sources
• Outputs SDK/toolchain into tarball or shell-wrapped installer
• Multilib toolchain/SDK support
  – Multiple arch-optimized libraries in one SDK
  – Started in master after Danny by Mark Hatle
  – Lots of fixes and recipe updates recently
Reuse It For Builds

• Provide `external-<name>-toolchain.bb`
• `TCMODE = “external-<name>”`
• `TCLIBC = “external-<name>-toolchain”`

• May require adjusting toolchain’s internal directory structure, native vs. target sysroots for `external-<name>-toolchain “scrapping” recipes to work`
  – Otherwise update `external-<name>-toolchain` for new structure
Reuse It For SDK

- Provide `external-<name>-sdk-toolchain.bb`
- Refer to earlier slides “Packaging SDK, Config and Recipe”
- All the needed toolchain binaries will be installed and packaged in the host side of SDK
- While target libraries and headers will be in the target side of SDK
- If done properly, the resulting SDK (even if contains other pieces, like QtE) can be fed back to the next Yocto build, using same `external-*-toolchain` recipes!
Toolchain-less SDK 1/2

- SDK contains extra libraries and header files for the target, as well as additional host tools
  - Excludes cross-toolchain components, such as gcc, gdb, binutils or glibc
- Toolchain components are expected to be provided separately for SDK to be useful
- Helpful when cannot distribute 3rd party binary toolchain with own SDK
- SDK customers are advised to acquire their copy of the toolchain from the source
Toolchain-less SDK 2/2

- populate_sdk() changes for glibc removal:

  ```
  opkg_dir = "${SDK_OUTPUT}/${SDKTARGETSYSROOT}"
  opkg-cl -o ${opkg_dir} -f ${opkg_dir}/etc/opkg.conf \ 
    --force-depends remove ${TOOLCHAIN_TARGET_EXCLUDE}
  find ${SDK_OUTPUT} -depth -type d -empty -print0 | xargs \ 
    -r0 /bin/rmdir
  ```

- Was in Classic-OE, need to submit to OE-Core
Canadian Cross Overview

- The Canadian Cross is a technique for building cross compilers for other machines.
- Given three machines A, B, and C, one uses machine A (e.g. running modern Linux distro on a 64bit x86) to build a cross compiler that runs on machine B (e.g. running older Linux distro on a 32bit x86) to create executables for machine C (e.g. running Poky/Angstrom/Arago on an ARM).
- The term Canadian Cross came about because at the time that these issues were under discussion, Canada had three national political parties.
Canadian Cross in Yocto

• **cross-canadian.bbclass**
  - SDKMACHINE **vs.** MACHINE

• **Output:**
  - gcc-cross-canadian-${TRANSLATED_TARGET_ARCH}
  - gdb-cross-canadian-${TRANSLATED_TARGET_ARCH}
  - binutils-cross-canadian-${TRANSLATED_TARGET_ARCH}

• “crosssdk” toolchain
• “nativesdk” tools
• Self-contained binaries
Self-contained Binaries

- Benefits of pre-determined system dependencies
  - Dynamic loader/interpreter
  - Dynamic libraries
- ELF headers
  - PT_INTERP section
  - RPATH/RUNPATH
- chrpath – provided by host system
  - Limitations – cannot grow fields/sections
- PatchELF is a better alternative
  - seen as extra build dependency
Relocatability in Denzil

• None
  – SDK path is hardcoded in binaries

• Custom solution with shell stubs in Arago
  – Each shell stub has actual name of binary
  – Binary is renamed to have .real suffix
  – Stub prepends `LD_LIBRARY_PATH` with path to our SDK libraries
  – Executes the .real binary via our SDK ld-linux.so.2 loader, passing all parameters
Relocatability in Danny+

- Shell-wrapped installer to adjust ELF headers
  - Calls `relocate_sdk.py` script on install and removes it
- Python script directly pokes into binaries to adjust ELF headers
  - Not very flexible – cannot expand fields, requires original paths to be long enough upfront
- Binaries use `$ORIGIN` for locating dynamic libraries, relative to binaries

- PatchELF tool is a better alternative to the above script, but seen as extra build dependency
Canadian with Regular Cross

• Mixing binaries:
  – Self-contained binaries with own dynamic loader and libraries
  – Regular binaries relying on host system loader and libraries

• Care needed when adjusting ELF headers
  – Regular binaries depend on host system dynamic loader (ld-linux.so) and libraries (libc.so)
  – Updating their ELF headers may corrupt them

• In Danny, `relocate_sdk.py` will mangle all binaries
  – needed custom filter in Arago

• In master, generic logic was added to skip such binaries
Thank you

Q&A