Activity Five

User-space: Advanced Topics

Rudi Streif

What We Are Going To Do

- Most of your development work will likely be developing your own software packages, building them with the Yocto Project and installing them into a root file system built with the Yocto Project.
- Let’s look at some typical tasks beyond creating the base recipe:
  - Customizing Packaging
  - Package Installation Scripts
  - System Services

Activity Setup

- Initialize the Build Environment
  - `cd /scratch/working`
  - `source ../poky/oe-init-build-env build-userspace`
- Adjust Configuration
  - `vi conf/local.conf`

```
MACHINE = "qemux86-64"
DL_DIR ?= "/scratch/downloads"
SSTATE_DIR ?= "/scratch/sstate-cache"
EXTRA_IMAGE_FEATURES ?= "debug-tweaks dbg-pkgs dev-pkgs package-management"
```

- Build
  - `bitbake -k core-image-minimal`
- Test
  - `runqemu qemux86-64 nographic`

Activity Setup - Continued

- Create Layer
  - `devtool create-workspace meta-uspapps`
- Copy Source Files
  - `cd ..`
  - `cp -r /scratch/src/userspace/uspsrc .`

Packaging

- Packaging is the process of putting artifacts from the build output into one or more packages for installation by a package management system.
- Packaging is performed by the package management classes:
  - `package_rpm` – RPM style packages
  - `package_deb` – Debian style packages
  - `package_ipk` – IPK package files used by the OPK package manager
- You configure the package management in `conf/local.conf`:

```
PACKAGE_CLASSES ?= "package_rpm"
```

  - You can add more than one of the package classes.
  - Only the first one will be used to create the root file system.
  - However, this does not install the package manager itself.
- Install the package manager in `conf/local.conf`:

```
EXTRA_IMAGE_FEATURES ?= "package-management"
```

Package Splitting

- Packaging Splitting is the process of putting artifacts from the build output into different packages.
- Package splitting allows you to select what you need to control the footprint of your root file system.
- Package splitting is controlled by the variables:
  - `PACKAGES` – list of package names, default:

```
PACKAGES = "${PN}-dbg ${PN}-staticdev ${PN}-dev ${PN}-doc \
   ${PN}-locale ${PACKAGE_BEFORE_PN} ${PN}"
```

  - `FILES` – list of directories and files that belong into the package:

```
SOLIBS = "*.so.*“
FILES_${PN} = "${bindir}/* ${sbindir}/* ${libexecdir}/* \
   ${libdir}/lib* {SOLIBS} ${sysconfdir} ${sharedstatedir} \
   ${localstatedir} ${base_bindir}/* ${base_sbindir}/* \
   ${base_libdir}/*${SOLIBS} ${base_prefix}/lib/udev/rules.d \
   ${prefix}/lib/udev/rules.d ${datadir}/${BPN}\
   ${libdir}/${BPN}/* ${datadir}/pixmaps \
   ${datadir}/applications ${datadir}/idl ${datadir}/omf \
   ${datadir}/sounds ${libdir}/bonobo/servers"
```

Package Splitting - Continued

- The package classes process the `PACKAGES` list from left to right, producing the `${PN}-dbg` package first and the `${PN}` package last.
- The order is important, since a package consumes the files that are associated with it.
- The `${PN}` package is pretty much the “kitchen sink”: it consumes all standard leftover artifacts.
- BitBake syntax only allows prepending (+=) or appending (=+) to variables:
  - Prepend `PACKAGES` – place standard artifacts into different packages
  - Append `PACKAGES` – place any leftover packages in non-standard installation directories those packages.
- The variable `PACKAGE_BEFORE_PN` allows you to insert packages right before the `${PN}` package is created.

Packaging QA

- The insane class adds plausibility and error checking to the packaging process.
- You can find a list of the checks in the Reference Manual: http://www.yoctoproject.org/docs/2.3/ref-manual/ref-manual.html#ref-classes-insane
- Some of the more common ones:
  - `already-stripped` – debug symbols already stripped
  - `installed-vs-shipped` – checks for artifacts that have not been packaged
  - `ldflags` – checks if `LDFLAGS` for cross-linking has been passed
  - `packages-list` – same package has been listed multiple times in `PACKAGES`
- Sometimes the checks can get into your way…
  - `INSANE_SKIP_<packagename> += "<check>"`
  - Skips <check> for <packagename>.

Example – The Fibonacci Library

- Source code in /scratch/working/uspsrc/fibonacci-lib
  - Builds static and dynamic libraries to calculate the Fibonacci series and an application to test it.
- Create development environment
  - `cd /scratch/working/build-userspace`
  - `devtool add fibonacci-lib /scratch/working/uspsrc/fibonacci-lib`
- Build the recipe
  - `bitbake fibonacci-lib`
- Add to your image (conf/local.conf):

```
IMAGE_INSTALL_append = " fibonacci-lib"
```

- Build and test image
  - `bitbake core-image-minimal`
  - `runqemu qemux86-64 nographic`

Example – The Fibonacci Library (continued)

- Edit the recipe `meta-uspapps/recipes/fibonacci-lib/fibonacci-lib.bb` and place the `fibonacci` test application into its own package `${PN}-examples`

```
PACKAGE_BEFORE_PN = "${PN}-examples"
FILES_${PN}-examples = "${bindir}/fibonacci"
```

- Add to your image (`conf/local.conf`):

```
IMAGE_INSTALL_append = " fibonacci-lib fibonacci-lib-examples"
```

- Build and test image
  - `bitbake core-image-minimal`
  - `runqemu qemux86-64 nographic`

Package Installation Scripts

- Package management systems have the ability to run scripts before and after a package is installed, upgraded, or removed.
- These are typically shell scripts and they can be provided by the recipe using these variables:
  - `pkg_preinst_<packagename>`: Preinstallation script that is run *before the package is installed*.
  - `pkg_postinst_<packagename>`: Postinstallation script that is run *after the package is installed*.
  - `pkg_prerm_<packagename>`: Pre-uninstallation script that is run *before the package is uninstalled*.
  - `pkg_postrm_<packagename>`: Post-uninstallation script that is run *after the package is uninstalled*.

```
pkg_postinst_${PN}() {
#!/bin/sh
# shell commands go here
}
```

Script Skeleton

```
pkg_postinst_${PN}() {
#!/bin/sh
if [ x"$D" = "x" ]; then
   # target execution
else
   # build system execution
fi
}
```

Conditional Execution

Example – Conditionally running ldconfig

- The Fibonacci library installs a dynamic library `libfibonacci.so.1.0` on the target system in `/usr/lib`.
- For `ld` to be able to locate the library it must be added to the ld cache and its symbolic name (soname) must be linked. That is done by running `ldconfig` on the target.
- Add a post installation script to the `${PN}` package that only runs `ldconfig` when it is run on the target but not when the build system creates the root file system.

```
pkg_postinst_${PN}() {
#!/bin/sh
if [ x"$D" = "x" ]; then
   # target execution
   ldconfig
   exit 0
else
   # build system execution
   exit 1
fi
}
```

Installation for Packaging

- Makefile Installation

```
INSTALL ?= install
.PHONY: install
Install:
    $(INSTALL) -d $(DESTDIR)/usr/bin
    $(INSTALL) -m 0755 $(TARGET) $(DESTDIR)/usr/bin
```

- Recipe Installation
  - Providing/overriding the do_install task

```
do_install() {
   install –d ${D}${bindir}
   install –m 0755 ${B}/bin/* ${D}{bindir}
}
```

  - The build system defines a series of variables for convenience:

bindir = “/usr/bin”

sbindir = “/usr/sbin”

libdir = “/usr/lib”

libexecdir = “/usr/lib”

sysconfdir = “/etc/”

datadir = “/usr/share”

mandir = “/use/share/man”

includedir = “/usr/include”

Debugging Packaging

- Check the packaging logfiles in `${WORKDIR}/temp`
- Check installation of artifacts in `${WORKDIR}/image`
  - The do_install task installs the artifacts into this directory.
  - If artifacts are missing they are packaged.
- Check packaging artifacts in `${WORKDIR}/package`
  - This where the artifacts are staged for packaging, including the ones created for the debug packages.
- Check package splitting in `${WORKDIR}/packages-split`
  - Packages and their content are staged here by package name before they are wrapped by the package manager.
  - Allows you to verify if the artifacts have indeed been placed into the correc package.
- Check created packages in `${WORKDIR}/deploy-<pkgmgr>`

Package Architecture

- The build system distinguishes packages by their hardware dependencies into three main categories:
  - Tune – Generic CPU architecture such as core2_32, corei7, armv7, etc. This is the default and typically appropriate for userspace packages.
  - Machine – Specific machine architecture. Appropriate for packages that require particular hardware features of a machine. Typically applicable to kernel, drivers, and bootloader.
  - All – Package applies to all architectures such as shell scripts, managed runtime code (Python, Lua, Java, …), configuration files, etc.
- Package architecture is controlled by the `PACKAGE_ARCH` variable:
  - Tune (default) – `PACKAGE_ARCH = "${TUNE_PKGARCH}"`
  - Machine – `PACKAGE_ARCH = "${MACHINE_ARCH}"`
  - All – `inherit allarch`
- Note: Package architecture does not simply determine into what category a package is placed but determines compiler and linker flags and other build options.

System Services

- If your software package is a system service that eventually needs to be started when the system boots you need to add the scripts and service files.
- **SysVInit**
  - Inherit `update-rc.d` class.
  - `INITSCRIPT_PACKAGES` - List of packages that contain the init scripts for this software package. This variable is optional and defaults to `INITSCRIPT_PACKAGES = "${PN}"`.
  - `INITSCRIPT_NAME` - The name of the init script.
  - `INITSCRIPT_PARAMS` - The parameters passed to `update-rc.d`. This can be a string such as `"defaults 80 20"` to start the service when entering run levels 2, 3, 4, and 5 and stop it from entering run levels 0, 1, and 6.
- **Systemd**
  - Inherit `systemd` class.
  - `SYSTEMD_PACKAGES` - List of packages that contain the systemd service files for the software package. This variable is optional and defaults to `SYSTEMD_PACKAGES = "${PN}"`.
  - `SYSTEMD_SERVICE` - The name of the service file.

Example – The Fibonacci Server

- Source code in /scratch/working/uspsrc/fibonacci-srv
  - Builds a TCP socket server listening on port 9999 for the number of terms and responds with the list of Fibonacci terms.
- Create development environment
  - `cd /scratch/working/build-userspace`
  - `devtool add fibonacci-srv /scratch/working/uspsrc/fibonacci-srv`
- Add system service startup to the recipe

  `meta-uspapps/recipes/fibonacci-srv/fibonacci-srv.bb`

```
inherit update-rc.d systemd
INITSCRIPT_NAME = "fibonacci-srv"
INITSCRIPT_PARAMS = "start 99 3 5 . stop 20 0 1 2 6 ."
SYSTEMD_SERVICE = "fibonacci-srv.service"
```

- Build the recipe
  - `bitbake fibonacci-srv`
- Add to your image (conf/local.conf):

```
IMAGE_INSTALL_append = " fibonacci-srv"
```

- Build and test image
  - `bitbake core-image-minimal`
  - `runqemu qemux86-64`
  - `nc localhost 9999`

Changing the System Manager

- SysVInit is the default system manager for the Poky distribution.
- To use systemd add it to your `conf/local.conf` file, or better, to your distribution configuration:

```
DISTRO_FEATURES_append = " systemd"

VIRTUAL-RUNTIME_init_manager = "systemd"
```

- If you exclusively want to use systemd, you can remove SysVInit from you root file system image with:

```
DISTRO_FEATURES_BACKFILL_CONSIDERED = "sysvinit"

VIRTUAL-RUNTIME_initscripts = ""
```