Shrinkwrap

Arm Limited

May 03, 2024

CONTENTS

CHAPTER

ONE

CONTENTS

1.1 Overview

1.1.1 Introduction

Shrinkwrap is a tool to simplify the process of building and running firmware on Arm Fixed Virtual Platforms (FVP). It provides a number of configurations that can be used out-of-the-box as well as enabling users to compose and extend them in a manner that can be easily shared and reused. No more shall engineers have to fight with inexplicable fragments of hand-me-down bash code or endlessly search for FVP command line parameters!

Shrinkwrap focuses on building FW stacks and configuring the FVP for a desired set of architecture features so that all components are consistent. Engineers bring their own kernel and rootfs to run on top of this foundation.

Shrinkwrap provides an intuative command line interface frontend and (by default) a container-based backend so users don't have to think about the tools required to build or run their configs. Everything is also transparent; users can discover every single invoked command with the `--dry-run` option.

Configs are defined in YAML and can easily be composed and extended using the built-in layering system.

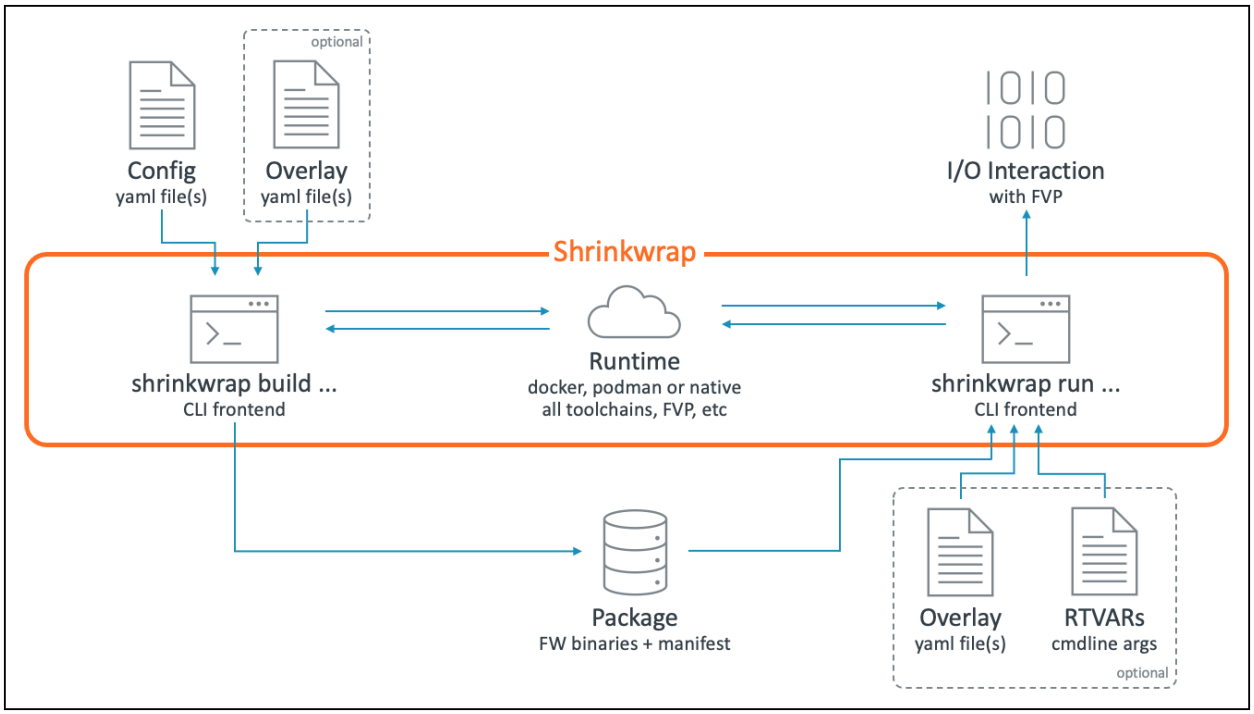
See *Quick Start Guide* to get up and running.

1.1.2 Features

- A simple and intuitive command line interface enables:
  - Acquire and build all required firmware components for a given configuration
  - Package built firmware components for easy distribution
  - Configure and boot the FVP with the packaged firmware components
- Introspect and use any of the supplied the out-of-box configurations
- Create your own configurations by composing with and extending others
- Choose from Docker or Podman runtime backends or run everything natively if you prefer
- Ensure Reproducible builds with supplied runtime container images
- Transparently view the generated bash commands for a given config build or run
- Parallelize builds to make best use of available resources
- Acquire source from Git remote or point to existing Git local repo
- Easily switch between all Arm architecture extensions v8.0 - v9.x.

1.1.3 Architecture

Shrinkwrap is implemented in Python and has a command line interface similar to git, with sub-commands that take options. The Python code parses the supplied config(s) to generate shell commands that are executed in a backend runtime. The runtime is specified by the user and may be `null` (executed natively on the user's system), or a container runtime such as `docker` or `podman`. For the container runtimes, a standard image is provided with all tools preinstalled.



The command line interface has 2 main commands; build and run. The build command takes a configuration (a yaml file) along with an optional overlay which tweaks settings in the config, then builds and packages all the components described within. The package could optionally be distributed for use by others. The run command launches the built package on the FVP, allowing the user to interact with it. The definition of how to run the FVP is contained in the original config that was supplied at build time and is included in the package. The run command allows further tweaking of this runtime config with another optional overlay. The user can also optionally provide values for any runtime variables (RTVARs) that are defined as part of the config. These are typically used to point to the kernel or rootfs that should be used.

1.1.4 Repository Structure

Directory	Description
./config	Shrinkwrap standard config store.
./docker	Scripts to generate docker images used by shrinkwrap's container runtimes.
./documentation	Source for this documentation.
./shrinkwrap	Shrinkwrap Python tool implementation.
./test	Automated tests.

1.1.5 Repository License

The software is provided under an MIT license (more details in *License*).

Contributions to the project should follow the same license.

1.1.6 Contributions and Bug Reports

Contributions are accepted under the MIT license. Only submit contributions where you have authored all of the code.

If you're hitting an error/bug and need help, it's best to raise an issue in GitLab.

1.1.7 Maintainer(s)

- Ryan Roberts <ryan.roberts@arm.com>

1.2 User Guide

1.2.1 Quick Start Guide

Install Shrinkwrap

Packages don't yet exist, so currently the only way to install Shrinkwrap is to install its dependencies and clone the git repository.

Shrinkwrap is tested on **Ubuntu 20.04** although other Linux distributions are likely to JustWork (TM). macOS is also known to work when using the docker runtime as long as Docker Desktop has first been installed.

Shrinkwrap requires **at least Python 3.6.9**. Older versions may work, but are not tested.

```
sudo apt-get install git netcat-openbsd python3 python3-pip telnet
sudo pip3 install pyyaml termcolor tuxmake
git clone https://git.gitlab.arm.com/tooling/shrinkwrap.git
export PATH=$PWD/shrinkwrap/shrinkwrap:$PATH
```

If using a Python version older than 3.9, you will also need to install the `graphlib-backport` pip package:

```
sudo pip3 install graphlib-backport
```

If using Docker Runtime Backend

If Docker was not previously set up on your system, you will need to install the package, create a 'docker' group and add your user to it. This allows shrinkwrap to interact with docker without needing sudo. For more information see docker linux-postinstall.

```
sudo apt-get install docker.io
sudo groupadd docker
sudo usermod -aG docker $USER
# Log out/log in for change to take effect
```

If using Podman Runtime Backend

Note: Podman is only available within Ubuntu repositories from Ubuntu 20.10 and newer. See podman installation instructions for installation methods for other distributions.

```
sudo apt-get install podman
```

Optional Environment Variables

Shrinkwrap consumes the following set of optional environment variables:

name	default	description
SHRINKWRAP_CONFIG	<None>	Colon-separated list of paths to config stores. Configs are searched for relative to the current directory as well as relative to these paths.
SHRINKWRAP_BUILD	~/.shrinkwrap/build	Location where config builds are performed. Each config has its own subdirectory, with further subdirectories for each of its components.
SHRINKWRAP_PACKAGE	~/.shrinkwrap/package	Location where config builds are packaged to. When running a config, it is done from the package location.

Guided Tour: Configure a platform and boot a kernel

This section provides a guided tour of Shrinkwrap, using a common use case of building required platform FW and configuring the FVP for Armv9.3 and booting a kernel. This example uses EDK2 (UEFI) but many other options are available.

Note: By default, the below commands will use the docker runtime and automatically download and use the appropriate container image from Docker Hub. Alternatively, you can choose to run with the `null` runtime by providing `--runtime=null` (between `shrinkwrap` and the sub-command). This will cause all commands to be executed on the native system. Users are responsible for setting up the environment in this case. Or if you have chosen to use Podman as the runtime backend, add `--runtime=podman`.

First invoke the tool to view help:

```
shrinkwrap --help
shrinkwrap <command> --help
```

Now, inspect the available configs:

```
shrinkwrap inspect
```

This will show all of the (concrete) configs in the config store. The below output shows a sample. Notice that each config lists its runtime variables ("rtvars") along with their default values. `None` means there is no default and the user must provide a value when running the config. (A "concrete" config is one that is deemed ready-to-use out-of-the-box. Whereas a config "fragment" is a piece of config that is usually composed with others and configured into a concrete config. You can view non-concrete fragments by providing extra args).

```
name:                bootwrapper.yaml

description:         Best choice for: I have a linux-system.axf boot-wrapper and
                     want to run it.

                     This config does not build any components (although
                     shrinkwrap still requires you to build it before running).
                     Instead the user is expected to provide a boot-wrapper
                     executable (usually called linux-system.axf) as the
                     BOOTWRAPPER rtvar, which will be executed in the FVP. A
                     ROOTFS can be optionally provided. If present it is loaded
                     into the virtio block device (/dev/vda).

concrete:            True

run-time variables:  LOCAL_NET_PORT:         8022
                     BOOTWRAPPER:            None
                     ROOTFS:

--------------------------------------------------------------------------------

name:                cca-3world.yaml

description:         Brings together a software stack to demonstrate Arm CCA
                     running on FVP in a three-world configuration. Includes
                     TF-A in root world, RMM in realm world, and Linux in Normal
                     world.

                     In order to launch realm VMs, the user must bring their own
                     rootfs that contains a realm-aware kvmtool and an RSI-aware
                     guest kernel image.

concrete:            True

run-time variables:  LOCAL_NET_PORT:         8022
                     BL1:                    ${artifact:BL1}
                     FIP:                    ${artifact:FIP}
                     KERNEL:                 ${artifact:KERNEL}
                     ROOTFS:

--------------------------------------------------------------------------------

name:                cca-4world.yaml

description:         Brings together a software stack to demonstrate Arm CCA
                     running on FVP in a four-world configuration. Includes TF-A
                     in root world, Hafnium and some demo secure partitions in
                     secure world, RMM in realm world, and Linux in Normal
                     world.

                     In order to launch realm VMs, the user must bring their own
                     rootfs that contains a realm-aware kvmtool and an RSI-aware
                     guest kernel image.
```

(continues on next page)

(continued from previous page)

```
concrete:            True

run-time variables:  LOCAL_NET_PORT:         8022
                     BL1:                    ${artifact:BL1}
                     FIP:                    ${artifact:FIP}
                     KERNEL:                 ${artifact:KERNEL}
                     ROOTFS:

--------------------------------------------------------------------------------

name:                ffa-tftf.yaml

description:         Brings together a software stack to demonstrate Arm FF-A
                     running on FVP. Includes TF-A in secure EL3, Hafnium in
                     secure EL2 and some demo TF-A test secure partitions.

concrete:            True

run-time variables:  LOCAL_NET_PORT:         8022
                     BL1:                    ${artifact:BL1}
                     FIP:                    ${artifact:FIP}
                     DTB:                    ${artifact:DTB}
                     CMDLINE:                console=ttyAMA0
                                             earlycon=pl011,0x1c090000
                                             root=/dev/vda ip=dhcp
                     KERNEL:                 None
                     ROOTFS:
                     EDK2FLASH:              ${artifact:EDK2FLASH}

--------------------------------------------------------------------------------

name:                ns-edk2.yaml

description:         Best choice for: I want to run Linux on FVP, booting with
                     ACPI/DT, and have easy control over its command line.

                     Brings together TF-A and EDK2 to provide a simple non-
                     secure world environment running on FVP. Allows easy
                     specification of the kernel image and command line, and
                     rootfs at runtime (see rtvars). ACPI is provided by UEFI.

                     An extra rtvar is added (DTB) which allows specification of
                     a custom device tree. By default (if not overriding the
                     rtvar), the upstream kernel device tree is used. DT is
                     enabled by default. Use 'acpi=force' to enable ACPI boot.

                     By default (if not overriding the rtvars) a sensible
                     command line is used that will set up the console for
                     logging and attempt to mount the rootfs image from the
                     FVP's virtio block device. However the default rootfs image
                     is empty, so the kernel will panic when attempting to
```

(continues on next page)

(continued from previous page)

```
                    mount; the user must supply a rootfs if it is required that
                    the kernel completes its boot. No default kernel image is
                    supplied and the config will refuse to run unless it is
                    explicitly specified.

                    Note that by default, a pre-canned flash image is loaded
                    into the model, which contains UEFI variables directing
                    EDK2 to boot to the shell. This will cause startup.nsh to
                    be executed and will start the kernel boot. This way
                    everything is automatic. By default, all EDK2 output is
                    muxed to stdout. If you prefer booting UEFI to its UI,
                    override the EDK2FLASH rtvar with an empty string and
                    override terminals.'bp.terminal_0'.type to 'telnet'.

concrete:           True

run-time variables: LOCAL_NET_PORT:         8022
                    BL1:                    ${artifact:BL1}
                    FIP:                    ${artifact:FIP}
                    DTB:                    ${artifact:DTB}
                    CMDLINE:                console=ttyAMA0
                                            earlycon=pl011,0x1c090000
                                            root=/dev/vda ip=dhcp
                    KERNEL:                 None
                    ROOTFS:
                    EDK2FLASH:              ${artifact:EDK2FLASH}

--------------------------------------------------------------------------------

name:               ns-preload.yaml

description:        Best choice for: I just want to run Linux on FVP.

                    A simple, non-secure-only configuration where all
                    components are preloaded into memory (TF-A's BL31, DTB and
                    kernel). The system resets directly to BL31. Allows easy
                    specification of a custom command line at build-time (via
                    build.dt.params dictionary) and specification of the device
                    tree, kernel image and rootfs at run-time (see rtvars).

                    By default (if not overriding the rtvars), the upstream
                    kernel device tree is used along with a sensible command
                    line that will set up the console for logging and attempt
                    to mount the rootfs image from the FVP's virtio block
                    device. However the default rootfs image is empty, so the
                    kernel will panic when attempting to mount; the user must
                    supply a rootfs if it is required that the kernel completes
                    its boot. No default kernel image is supplied and the
                    config will refuse to run unless it is explicitly
                    specified.  Note: If specifying a custom dtb at runtime,
                    this will also override any command line specified at build
                    time, since the command line is added to the chosen node of
```

(continues on next page)

(continued from previous page)

```
                    the default dtb.

concrete:           True

run-time variables: LOCAL_NET_PORT:         8022
                    BL31:                   ${artifact:BL31}
                    DTB:                    ${artifact:DTB}
                    KERNEL:                 None
                    ROOTFS:
```

Now build the `ns-edk2.yaml` config. This allows booting a kernel on FVP, using edk2 as the bootloader (it uses DT by default, but can be made to use ACPI by passing `acpi=force` at runtime). (optionally add `--verbose` to see all the output from the component build systems).

```
shrinkwrap build --overlay=arch/v9.3.yaml ns-edk2.yaml
```

This will sync all the required repos, build the components and package the artifacts.

Warning: By default, Shrinkwrap will sync all component repos to the revision specified in the config on every build invocation. If you have made changes in the working directory, your CHANGES WILL BE LOST! You can override this behaviour so that Shrinkwrap just builds whatever is in the working directory by adding `--no-sync <component>` or `--no-sync-all` to the command line.

Alternatively, pass `--dry-run` to view the shell script that would have been run:

```
shrinkwrap build --overlay=arch/v9.3.yaml --dry-run ns-edk2.yaml
```

```
#!/bin/bash
# SHRINKWRAP AUTOGENERATED SCRIPT.

# Exit on error.
set -e

# Remove old package.
rm -rf /data_nvme0n1/ryarob01/shrinkwrap_demo/live/package/ns-edk2.yaml > /dev/null 2>&1␣
↪|| true
rm -rf /data_nvme0n1/ryarob01/shrinkwrap_demo/live/package/ns-edk2 > /dev/null 2>&1 ||␣
↪true

# Create directory structure.
mkdir -p /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2/dt
mkdir -p /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2/edk2
mkdir -p /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2/edk2flash
mkdir -p /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2/tfa
mkdir -p /data_nvme0n1/ryarob01/shrinkwrap_demo/live/package/ns-edk2

# Sync git repo for config=ns-edk2 component=dt.
pushd /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2
if [ ! -d "dt/.git" ] || [ -f "./.dt_sync" ]; then
      rm -rf dt > /dev/null 2>&1 || true
      mkdir -p .
```

(continues on next page)

(continued from previous page)

```
      touch ./.dt_sync
      git clone --quiet git://git.kernel.org/pub/scm/linux/kernel/git/devicetree/
↪devicetree-rebasing.git dt
      pushd dt
      git checkout --quiet --force v6.1-dts
      git submodule --quiet update --init --checkout --recursive --force
      popd
      rm ./.dt_sync
fi
popd

# Sync git repo for config=ns-edk2 component=edk2.
pushd /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2
if [ ! -d "edk2/edk2/.git" ] || [ -f "edk2/.edk2_sync" ]; then
      rm -rf edk2/edk2 > /dev/null 2>&1 || true
      mkdir -p edk2
      touch edk2/.edk2_sync
      git clone --quiet https://github.com/tianocore/edk2.git edk2/edk2
      pushd edk2/edk2
      git checkout --quiet --force edk2-stable202211
      git submodule --quiet update --init --checkout --recursive --force
      popd
      rm edk2/.edk2_sync
fi
if [ ! -d "edk2/edk2-platforms/.git" ] || [ -f "edk2/.edk2-platforms_sync" ]; then
      rm -rf edk2/edk2-platforms > /dev/null 2>&1 || true
      mkdir -p edk2
      touch edk2/.edk2-platforms_sync
      git clone --quiet https://github.com/tianocore/edk2-platforms.git edk2/edk2-
↪platforms
      pushd edk2/edk2-platforms
      git checkout --quiet --force 20e07099d8f11889d101dd710ca85001be20e179
      git submodule --quiet update --init --checkout --recursive --force
      popd
      rm edk2/.edk2-platforms_sync
fi
if [ ! -d "edk2/acpica/.git" ] || [ -f "edk2/.acpica_sync" ]; then
      rm -rf edk2/acpica > /dev/null 2>&1 || true
      mkdir -p edk2
      touch edk2/.acpica_sync
      git clone --quiet https://github.com/acpica/acpica.git edk2/acpica
      pushd edk2/acpica
      git checkout --quiet --force R10_20_22
      git submodule --quiet update --init --checkout --recursive --force
      popd
      rm edk2/.acpica_sync
fi
popd


# Sync git repo for config=ns-edk2 component=tfa.
pushd /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2
```

(continues on next page)

(continued from previous page)

```
if [ ! -d "tfa/.git" ] || [ -f "./.tfa_sync" ]; then
      rm -rf tfa > /dev/null 2>&1 || true
      mkdir -p .
      touch ./.tfa_sync
      git clone --quiet https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git tfa
      pushd tfa
      git checkout --quiet --force v2.8.0
      git submodule --quiet update --init --checkout --recursive --force
      popd
      rm ./.tfa_sync
fi
popd

# Build for config=ns-edk2 component=dt.
export CROSS_COMPILE=aarch64-none-elf-
pushd /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2/dt
DTS=fvp-base-revc.dts
INITRD_START=
INITRD_END=
DT_BASENAME=$(basename ${DTS} .dts)
DTB_INTER=src/arm64/arm/${DT_BASENAME}.dtb
DTB_FINAL=/data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-edk2/dt/dt_bootargs.
↪dtb
make CPP=${CROSS_COMPILE}cpp -j28 ${DTB_INTER}
CHOSEN=
if [ ! -z "" ]; then
CHOSEN="${CHOSEN}bootargs = \"\";\n"
fi
if [ ! -z "${INITRD_START}" ] && [ ! -z "${INITRD_END}" ]; then
INITRD_START_HI=$((${INITRD_START} >> 32) & 0xffffffff))
INITRD_START_LO=$((${INITRD_START} & 0xffffffff))
INITRD_END_HI=$((${INITRD_END} >> 32) & 0xffffffff))
INITRD_END_LO=$((${INITRD_END} & 0xffffffff))
CHOSEN="${CHOSEN}linux,initrd-start = <${INITRD_START_HI} ${INITRD_START_LO}>;\n"
CHOSEN="${CHOSEN}linux,initrd-end = <${INITRD_END_HI} ${INITRD_END_LO}>;\n"
fi
if [ -z "${CHOSEN}" ]; then
cp ${DTB_INTER} ${DTB_FINAL}
else
( dtc -q -O dts -I dtb ${DTB_INTER} ; echo -e "/ { chosen { ${CHOSEN} }; };" ) | dtc -q -
↪O dtb -o ${DTB_FINAL}
fi
if [ "${DTS}" = "fvp-base-revc.dts" ]; then
OVERLAY="/ {
  reserved-memory {
    fw: fw@7C000000 {
      reg = <0x00000000 0xFC000000 0 0x04000000>;
      no-map;
    };
  };
  timer {
    clock-frequency = <100000000>;
```

(continues on next page)

(continued from previous page)

```
  };
  psci {
    compatible = \"arm,psci-1.0\", \"arm,psci-0.2\";
    max-pwr-lvl = <2>;
  };
  cpus {
    cpu-map {
      cluster0 {
        core0 { cpu = <&{/cpus/cpu@0}>; };
        core1 { cpu = <&{/cpus/cpu@100}>; };
        core2 { cpu = <&{/cpus/cpu@200}>; };
        core3 { cpu = <&{/cpus/cpu@300}>; };
      };
      cluster1 {
        core0 { cpu = <&{/cpus/cpu@10000}>; };
        core1 { cpu = <&{/cpus/cpu@10100}>; };
        core2 { cpu = <&{/cpus/cpu@10200}>; };
        core3 { cpu = <&{/cpus/cpu@10300}>; };
      };
    };
  };
  bus@8000000 {
    motherboard-bus@8000000 {
      iofpga-bus@300000000 {
        virtio@200000 {
          status = \"okay\";
        };
      };
    };
  };
};"
( dtc -q -O dts -I dtb ${DTB_FINAL} ; echo -e "${OVERLAY}" ) | dtc -q -O dtb -o ${DTB_
↪FINAL}
fi
popd

# Build for config=ns-edk2 component=edk2.
export CROSS_COMPILE=aarch64-none-elf-
pushd /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2/edk2
export WORKSPACE=/data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2/edk2
export GCC5_AARCH64_PREFIX=$CROSS_COMPILE
export PACKAGES_PATH=$WORKSPACE/edk2:$WORKSPACE/edk2-platforms
export IASL_PREFIX=$WORKSPACE/acpica/generate/unix/bin/
export PYTHON_COMMAND=/usr/bin/python3
make -j28 -C acpica
source edk2/edksetup.sh
make -j28 -C edk2/BaseTools
build -n 28 -D EDK2_OUT_DIR=/data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-
↪edk2/edk2 -a AARCH64 -t GCC5 -p Platform/ARM/VExpressPkg/ArmVExpress-FVP-AArch64.dsc -
↪b RELEASE
popd
```

(continues on next page)

(continued from previous page)

```
# Build for config=ns-edk2 component=tfa.
export CROSS_COMPILE=aarch64-none-elf-
pushd /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/source/ns-edk2/tfa
make BUILD_BASE=/data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-edk2/tfa␣
↪PLAT=fvp DEBUG=0 LOG_LEVEL=40 ARM_DISABLE_TRUSTED_WDOG=1 FVP_HW_CONFIG_DTS=fdts/fvp-
↪base-gicv3-psci-1t.dts BL33=/data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-
↪edk2/edk2/RELEASE_GCC5/FV/FVP_AARCH64_EFI.fd ARM_ARCH_MINOR=5 ENABLE_SVE_FOR_NS=1␣
↪ENABLE_SVE_FOR_SWD=1 CTX_INCLUDE_PAUTH_REGS=1 BRANCH_PROTECTION=1 CTX_INCLUDE_MTE_
↪REGS=1 ENABLE_FEAT_HCX=1 CTX_INCLUDE_AARCH32_REGS=0 ENABLE_SME_FOR_NS=1 ENABLE_SME_FOR_
↪SWD=1 all fip
popd

# Copy artifacts for config=ns-edk2.
cp /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-edk2/dt/dt_bootargs.dtb /
↪data_nvme0n1/ryarob01/shrinkwrap_demo/live/package/ns-edk2/dt_bootargs.dtb
cp /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-edk2/edk2/RELEASE_GCC5/FV/
↪FVP_AARCH64_EFI.fd /data_nvme0n1/ryarob01/shrinkwrap_demo/live/package/ns-edk2/FVP_
↪AARCH64_EFI.fd
cp /data_nvme0n1/ryarob01/shrinkwrap_demo/shrinkwrap/config/edk2-flash.img /data_nvme0n1/
↪ryarob01/shrinkwrap_demo/live/package/ns-edk2/edk2-flash.img
cp /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-edk2/tfa/fvp/release/bl1.
↪bin /data_nvme0n1/ryarob01/shrinkwrap_demo/live/package/ns-edk2/bl1.bin
cp /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-edk2/tfa/fvp/release/bl2.
↪bin /data_nvme0n1/ryarob01/shrinkwrap_demo/live/package/ns-edk2/bl2.bin
cp /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-edk2/tfa/fvp/release/bl31.
↪bin /data_nvme0n1/ryarob01/shrinkwrap_demo/live/package/ns-edk2/bl31.bin
cp /data_nvme0n1/ryarob01/shrinkwrap_demo/live/build/build/ns-edk2/tfa/fvp/release/fip.
↪bin /data_nvme0n1/ryarob01/shrinkwrap_demo/live/package/ns-edk2/fip.bin
```

Now start the FVP. We will pass our own kernel and rootfs disk image as runtime variables. A config can define any number of runtime variables which may have default values (see `inspect` command above). If a variable has no default value, then the user must provide a value when invoking the `run` command. The `ns-edk2.yaml` config requires the user to provide a kernel, but the rootfs is optional. If the rootfs was omitted, the kernel would boot to the point where it attempts to mount the rootfs then panic (which is sufficient for some development use cases!).

```
shrinkwrap run --rtvar=KERNEL=path/to/Image --rtvar=ROOTFS=path/to/rootfs.img ns-edk2.
↪yaml
```

This starts the FVP and multiplexes all the UART terminals to stdout and forwards stdin to the `tfa+linux` uart terminal. This allows the user to interact directly with the FVP in a terminal without the need for a GUI setup:

```
[       fvp ] terminal_0: Listening for serial connection on port 5000
[       fvp ] terminal_1: Listening for serial connection on port 5001
[       fvp ] terminal_2: Listening for serial connection on port 5002
[       fvp ] terminal_3: Listening for serial connection on port 5003
[       fvp ]
[       fvp ] Info: FVP_Base_RevC_2xAEMvA: FVP_Base_RevC_2xAEMvA.bp.flashloader0:␣
↪FlashLoader: Loaded 100 kB from file '<root>/package/ns-preload/fip.bin'
[       fvp ]
[       fvp ] Info: FVP_Base_RevC_2xAEMvA: FVP_Base_RevC_2xAEMvA.bp.secureflashloader:␣
↪FlashLoader: Loaded 30 kB from file '<root>/package/ns-preload/bl1.bin'
```

(continues on next page)

(continued from previous page)

```
[       fvp ]
[       fvp ] libdbus-1.so.3: cannot open shared object file: No such file or directory
[       fvp ] libdbus-1.so.3: cannot open shared object file: No such file or directory
[ tfa+linux ] NOTICE:  BL31: v2.7(release):v2.7.0-391-g9dedc1ab2
[ tfa+linux ] NOTICE:  BL31: Built : 09:41:20, Sep 15 2022
[ tfa+linux ] INFO:    GICv3 with legacy support detected.
[ tfa+linux ] INFO:    ARM GICv3 driver initialized in EL3
[ tfa+linux ] INFO:    Maximum SPI INTID supported: 255
[ tfa+linux ] INFO:    Configuring TrustZone Controller
[ tfa+linux ] INFO:    Total 8 regions set.
[ tfa+linux ] INFO:    BL31: Initializing runtime services
[ tfa+linux ] INFO:    BL31: Preparing for EL3 exit to normal world
[ tfa+linux ] INFO:    Entry point address = 0x84000000
[ tfa+linux ] INFO:    SPSR = 0x3c9
[ tfa+linux ] [    0.000000] Booting Linux on physical CPU 0x0000000000 [0x410fd0f0]
[ tfa+linux ] [    0.000000] Linux version 5.15.0-rc2-gca9bfbea162d (ryarob01@e125769)␣
↪(aarch64-none-linux-gnu-gcc (GNU Toolchain for the A-profile Architecture 9.2-2019.12␣
↪(arm-9.10)) 9.2.1 20191025, GNU ld (GNU Toolchain for the A-profile Architecture 9.2-
↪2019.12 (arm-9.10)) 2.33.1.20191209) #1 SMP PREEMPT Thu Aug 4 11:31:55 BST 2022
[ tfa+linux ] [    0.000000] Machine model: FVP Base RevC
[ tfa+linux ] [    0.000000] earlycon: pl11 at MMIO 0x000000001c090000 (options '')
[ tfa+linux ] [    0.000000] printk: bootconsole [pl11] enabled
[ tfa+linux ] [    0.000000] efi: UEFI not found.
[ tfa+linux ] [    0.000000] Reserved memory: created DMA memory pool at␣
↪0x0000000018000000, size 8 MiB
[ tfa+linux ] [    0.000000] OF: reserved mem: initialized node vram@18000000,␣
↪compatible id shared-dma-pool
[ tfa+linux ] [    0.000000] NUMA: No NUMA configuration found
[ tfa+linux ] [    0.000000] NUMA: Faking a node at [mem 0x0000000080000000-
↪0x00000008fffffff]
[ tfa+linux ] [    0.000000] NUMA: NODE_DATA [mem 0x8ff7efc00-0x8ff7f1fff]
[ tfa+linux ] [    0.000000] Zone ranges:
[ tfa+linux ] [    0.000000]   DMA      [mem 0x0000000080000000-0x00000000ffffffff]
[ tfa+linux ] [    0.000000]   DMA32    empty
[ tfa+linux ] [    0.000000]   Normal   [mem 0x0000000100000000-0x00000008fffffff]
[ tfa+linux ] [    0.000000] Movable zone start for each node
[ tfa+linux ] [    0.000000] Early memory node ranges
[ tfa+linux ] [    0.000000]   node   0: [mem 0x0000000080000000-0x00000000ffffffff]
[ tfa+linux ] [    0.000000]   node   0: [mem 0x0000000880000000-0x00000008fffffff]
[ tfa+linux ] [    0.000000] Initmem setup node 0 [mem 0x0000000080000000-
↪0x00000008fffffff]
[ tfa+linux ] [    0.000000] cma: Reserved 32 MiB at 0x00000000fe000000
[ tfa+linux ] [    0.000000] psci: probing for conduit method from DT.
[ tfa+linux ] [    0.000000] psci: PSCIv1.1 detected in firmware.
[ tfa+linux ] [    0.000000] psci: Using standard PSCI v0.2 function IDs
[ tfa+linux ] [    0.000000] psci: MIGRATE_INFO_TYPE not supported.
[ tfa+linux ] [    0.000000] psci: SMC Calling Convention v1.2
...
```

Alternatively, you could have passed `--dry-run` to see the FVP invocation script:

```
shrinkwrap run --rtvar=KERNEL=path/to/Image --rtvar=ROOTFS=path/to/rootfs.img --dry-run␣
↪ns-edk2.yaml
```

(continues on next page)

(continued from previous page)

```
#!/bin/bash
# SHRINKWRAP AUTOGENERATED SCRIPT.

# Exit on error.
set -e

# Execute prerun commands.
SEMIHOSTDIR=`mktemp -d`
function finish { rm -rf $SEMIHOSTDIR; }
trap finish EXIT
cp ./path/to/Image ${SEMIHOSTDIR}/Image
cp <root>/package/ns-edk2/fvp-base-revc_args.dtb ${SEMIHOSTDIR}/fdt.dtb
cat <<EOF > ${SEMIHOSTDIR}/startup.nsh
Image dtb=fdt.dtb console=ttyAMA0 earlycon=pl011,0x1c090000 root=/dev/vda ip=dhcp
EOF

# Run the model.
FVP_Base_RevC-2xAEMvA \
    --plugin=$(which ScalableVectorExtension.so) \
    --stat \
    -C SVE.ScalableVectorExtension.has_sme2=1 \
    -C SVE.ScalableVectorExtension.has_sme=1 \
    -C SVE.ScalableVectorExtension.has_sve2=1 \
    -C bp.dram_metadata.is_enabled=1 \
    -C bp.dram_size=4 \
    -C bp.flashloader0.fname=<root>/package/ns-edk2/fip.bin \
    -C bp.flashloader1.fname=<root>/package/ns-edk2/edk2-flash.img \
    -C bp.hostbridge.userNetPorts=8022=22 \
    -C bp.hostbridge.userNetworking=1 \
    -C bp.refcounter.non_arch_start_at_default=1 \
    -C bp.refcounter.use_real_time=0 \
    -C bp.secure_memory=1 \
    -C bp.secureflashloader.fname=<root>/package/ns-edk2/bl1.bin \
    -C bp.smsc_91c111.enabled=1 \
    -C bp.terminal_0.mode=telnet \
    -C bp.terminal_0.start_telnet=0 \
    -C bp.terminal_1.mode=raw \
    -C bp.terminal_1.start_telnet=0 \
    -C bp.terminal_2.mode=raw \
    -C bp.terminal_2.start_telnet=0 \
    -C bp.terminal_3.mode=raw \
    -C bp.terminal_3.start_telnet=0 \
    -C bp.ve_sysregs.exit_on_shutdown=1 \
    -C bp.virtioblockdevice.image_path=./path/to/rootfs.img \
    -C bp.vis.disable_visualisation=1 \
    -C cache_state_modelled=0 \
    -C cluster0.NUM_CORES=4 \
    -C cluster0.PA_SIZE=48 \
    -C cluster0.check_memory_attributes=0 \
    -C cluster0.clear_reg_top_eret=2 \
```

(continues on next page)

(continued from previous page)

```
    -C cluster0.cpu0.semihosting-cwd=${SEMIHOSTDIR} \
    -C cluster0.ecv_support_level=2 \
    -C cluster0.enhanced_pac2_level=3 \
    -C cluster0.gicv3.cpuintf-mmap-access-level=2 \
    -C cluster0.gicv3.without-DS-support=1 \
    -C cluster0.gicv4.mask-virtual-interrupt=1 \
    -C cluster0.has_16k_granule=1 \
    -C cluster0.has_amu=1 \
    -C cluster0.has_arm_v8-1=1 \
    -C cluster0.has_arm_v8-2=1 \
    -C cluster0.has_arm_v8-3=1 \
    -C cluster0.has_arm_v8-4=1 \
    -C cluster0.has_arm_v8-5=1 \
    -C cluster0.has_arm_v8-6=1 \
    -C cluster0.has_arm_v8-7=1 \
    -C cluster0.has_arm_v8-8=1 \
    -C cluster0.has_arm_v9-0=1 \
    -C cluster0.has_arm_v9-1=1 \
    -C cluster0.has_arm_v9-2=1 \
    -C cluster0.has_arm_v9-3=1 \
    -C cluster0.has_branch_target_exception=1 \
    -C cluster0.has_brbe=1 \
    -C cluster0.has_brbe_v1p1=1 \
    -C cluster0.has_const_pac=1 \
    -C cluster0.has_hpmn0=1 \
    -C cluster0.has_large_system_ext=1 \
    -C cluster0.has_large_va=1 \
    -C cluster0.has_rndr=1 \
    -C cluster0.max_32bit_el=0 \
    -C cluster0.memory_tagging_support_level=3 \
    -C cluster0.pmb_idr_external_abort=1 \
    -C cluster0.stage12_tlb_size=1024 \
    -C cluster1.NUM_CORES=4 \
    -C cluster1.PA_SIZE=48 \
    -C cluster1.check_memory_attributes=0 \
    -C cluster1.clear_reg_top_eret=2 \
    -C cluster1.ecv_support_level=2 \
    -C cluster1.enhanced_pac2_level=3 \
    -C cluster1.gicv3.cpuintf-mmap-access-level=2 \
    -C cluster1.gicv3.without-DS-support=1 \
    -C cluster1.gicv4.mask-virtual-interrupt=1 \
    -C cluster1.has_16k_granule=1 \
    -C cluster1.has_amu=1 \
    -C cluster1.has_arm_v8-1=1 \
    -C cluster1.has_arm_v8-2=1 \
    -C cluster1.has_arm_v8-3=1 \
    -C cluster1.has_arm_v8-4=1 \
    -C cluster1.has_arm_v8-5=1 \
    -C cluster1.has_arm_v8-6=1 \
    -C cluster1.has_arm_v8-7=1 \
    -C cluster1.has_arm_v8-8=1 \
    -C cluster1.has_arm_v9-0=1 \
```

(continues on next page)

(continued from previous page)

```
    -C cluster1.has_arm_v9-1=1 \
    -C cluster1.has_arm_v9-2=1 \
    -C cluster1.has_arm_v9-3=1 \
    -C cluster1.has_branch_target_exception=1 \
    -C cluster1.has_brbe=1 \
    -C cluster1.has_brbe_v1p1=1 \
    -C cluster1.has_const_pac=1 \
    -C cluster1.has_hpmn0=1 \
    -C cluster1.has_large_system_ext=1 \
    -C cluster1.has_large_va=1 \
    -C cluster1.has_rndr=1 \
    -C cluster1.max_32bit_el=0 \
    -C cluster1.memory_tagging_support_level=3 \
    -C cluster1.pmb_idr_external_abort=1 \
    -C cluster1.stage12_tlb_size=1024 \
    -C pci.pci_smmuv3.mmu.SMMU_AIDR=2 \
    -C pci.pci_smmuv3.mmu.SMMU_IDR0=4592187 \
    -C pci.pci_smmuv3.mmu.SMMU_IDR1=6291458 \
    -C pci.pci_smmuv3.mmu.SMMU_IDR3=5908 \
    -C pci.pci_smmuv3.mmu.SMMU_IDR5=4294902901 \
    -C pci.pci_smmuv3.mmu.SMMU_ROOT_IDR0=3 \
    -C pci.pci_smmuv3.mmu.SMMU_ROOT_IIDR=1083 \
    -C pci.pci_smmuv3.mmu.SMMU_S_IDR1=2684354562 \
    -C pci.pci_smmuv3.mmu.SMMU_S_IDR2=0 \
    -C pci.pci_smmuv3.mmu.SMMU_S_IDR3=0 \
    -C pci.pci_smmuv3.mmu.root_register_page_offset=131072 \
    -C pctl.startup=0.0.0.0
```

Overlays are an important concept for Shrinkwrap. An overlay is a config fragment (either a yaml file or a json-encoded string) that can be passed separately on the command line and forms the top layer of the config. In this way, it can override or add any required configuration. You could achive the same effect by creating a new config and specifying the main config as a layer in that new config, but with an overlay, you can apply a config fragment to many different existing configs without the need to write a new config file each time. You can see overlays being using in the above commands to target a specific Arm architecture revision (v9.3 in the example). You can change the targetted architecture just by changing the overlay. There are many other places where overlays come in handy. See *Shrinkwrap Recipes* for more examples.

You will notice in the examples above, that only `build` commands include the overlay and `run` commands don't specify it. This is because the final config used for building is packaged in the built package, so when running the package, the presence of the overlay is implicit. However, a user could choose to provide an extra overlay at `run` time, that affects only the runtime portion to customize even further if desired.

For debug purposes, you can see a final, merged config by using the `process` command:

```
shrinkwrap process --action=merge --overlay=arch/v9.3.yaml ns-edk2.yaml
```

```
%YAML 1.2
---
name: ns-edk2
fullname: ns-edk2.yaml
description: 'Best choice for: I want to run Linux on FVP, booting with ACPI/DT, and
  have easy control over its command line.
```

(continues on next page)

(continued from previous page)

```
  Brings together TF-A and EDK2 to provide a simple non-secure world environment running
  on FVP. Allows easy specification of the kernel image and command line, and rootfs
  at runtime (see rtvars). ACPI is provided by UEFI.

  An extra rtvar is added (DTB) which allows specification of a custom device tree.
  By default (if not overriding the rtvar), the upstream kernel device tree is used.
  DT is enabled by default. Use ''acpi=force'' to enable ACPI boot.

  By default (if not overriding the rtvars) a sensible command line is used that will
  set up the console for logging and attempt to mount the rootfs image from the FVP''s
  virtio block device. However the default rootfs image is empty, so the kernel will
  panic when attempting to mount; the user must supply a rootfs if it is required
  that the kernel completes its boot. No default kernel image is supplied and the
  config will refuse to run unless it is explicitly specified.

  Note that by default, a pre-canned flash image is loaded into the model, which contains
  UEFI variables directing EDK2 to boot to the shell. This will cause startup.nsh
  to be executed and will start the kernel boot. This way everything is automatic.
  By default, all EDK2 output is muxed to stdout. If you prefer booting UEFI to its
  UI, override the EDK2FLASH rtvar with an empty string and override terminals.''bp.
↪terminal_0''.type
  to ''telnet''.'
concrete: true
graph: {}
build:
  dt:
    repo:
      .:
        remote: git://git.kernel.org/pub/scm/linux/kernel/git/devicetree/devicetree-
↪rebasing.git
        revision: v6.1-dts
    sourcedir: null
    builddir: null
    toolchain: aarch64-none-elf-
    params: {}
    prebuild:
    - DTS=fvp-base-revc.dts
    - INITRD_START=
    - INITRD_END=
    build:
    - DT_BASENAME=$$(basename $${DTS} .dts)
    - DTB_INTER=src/arm64/arm/$${DT_BASENAME}.dtb
    - DTB_FINAL=${param:builddir}/dt_bootargs.dtb
    - make CPP=$${CROSS_COMPILE}cpp -j${param:jobs} $${DTB_INTER}
    - CHOSEN=
    - if [ ! -z "${param:join_equal}" ]; then
    - CHOSEN="$${CHOSEN}bootargs = \"${param:join_equal}\";\n"
    - fi
    - if [ ! -z "$${INITRD_START}" ] && [ ! -z "$${INITRD_END}" ]; then
    - INITRD_START_HI=$$((($${INITRD_START} >> 32) & 0xffffffff))
    - INITRD_START_LO=$$(($${INITRD_START} & 0xffffffff))
    - INITRD_END_HI=$$((($${INITRD_END} >> 32) & 0xffffffff))
```

(continues on next page)

(continued from previous page)

```
    - INITRD_END_LO=$$(($${INITRD_END} & 0xffffffff))
    - CHOSEN="$${CHOSEN}linux,initrd-start = <$${INITRD_START_HI} $${INITRD_START_LO}>;\n
↪"
    - CHOSEN="$${CHOSEN}linux,initrd-end = <$${INITRD_END_HI} $${INITRD_END_LO}>;\n"
    - fi
    - if [ -z "$${CHOSEN}" ]; then
    - cp $${DTB_INTER} $${DTB_FINAL}
    - else
    - ( dtc -q -O dts -I dtb $${DTB_INTER} ; echo -e "/ { chosen { $${CHOSEN} }; };"
      ) | dtc -q -O dtb -o $${DTB_FINAL}
    - fi
    - if [ "$${DTS}" = "fvp-base-revc.dts" ]; then
    - "OVERLAY=\"/ {\n  reserved-memory {\n    fw: fw@7C000000 {\n      reg =
↪<0x00000000\
      \ 0xFC000000 0 0x04000000>;\n      no-map;\n    };\n  };\n  timer {\n    clock-
↪frequency\
      \ = <100000000>;\n  };\n  psci {\n    compatible = \\\"arm,psci-1.0\\\", \\\"\
      arm,psci-0.2\\\";\n    max-pwr-lvl = <2>;\n  };\n  cpus {\n    cpu-map {\n \
      \     cluster0 {\n        core0 { cpu = <&{/cpus/cpu@0}>; };\n        core1\
      \ { cpu = <&{/cpus/cpu@100}>; };\n        core2 { cpu = <&{/cpus/cpu@200}>;\
      \ };\n        core3 { cpu = <&{/cpus/cpu@300}>; };\n      };\n      cluster1\
      \ {\n        core0 { cpu = <&{/cpus/cpu@10000}>; };\n        core1 { cpu = <&{/
↪cpus/cpu@10100}>;\
      \ };\n        core2 { cpu = <&{/cpus/cpu@10200}>; };\n        core3 { cpu =\
      \ <&{/cpus/cpu@10300}>; };\n      };\n    };\n  };\n  bus@8000000 {\n    ␣
↪motherboard-bus@8000000\
      \ {\n      iofpga-bus@300000000 {\n        virtio@200000 {\n          status\
      \ = \\\"okay\\\";\n        };\n      };\n    };\n  };\n};\""
    - ( dtc -q -O dts -I dtb $${DTB_FINAL} ; echo -e "$${OVERLAY}" ) | dtc -q -O dtb
      -o $${DTB_FINAL}
    - fi
   postbuild: []
   artifacts:
     DTB: ${param:builddir}/dt_bootargs.dtb
 edk2:
   repo:
     edk2:
       remote: https://github.com/tianocore/edk2.git
       revision: edk2-stable202211
     edk2-platforms:
       remote: https://github.com/tianocore/edk2-platforms.git
       revision: 20e07099d8f11889d101dd710ca85001be20e179
     acpica:
       remote: https://github.com/acpica/acpica.git
       revision: R10_20_22
   sourcedir: null
   builddir: null
   toolchain: aarch64-none-elf-
   params:
     -a: AARCH64
     -t: GCC5
     -p: Platform/ARM/VExpressPkg/ArmVExpress-FVP-AArch64.dsc
```

(continues on next page)

(continued from previous page)

```
    -b: RELEASE
  prebuild:
  - export WORKSPACE=${param:sourcedir}
  - export GCC5_AARCH64_PREFIX=$$CROSS_COMPILE
  - export PACKAGES_PATH=$$WORKSPACE/edk2:$$WORKSPACE/edk2-platforms
  - export IASL_PREFIX=$$WORKSPACE/acpica/generate/unix/bin/
  - export PYTHON_COMMAND=/usr/bin/python3
  build:
  - make -j${param:jobs} -C acpica
  - source edk2/edksetup.sh
  - make -j${param:jobs} -C edk2/BaseTools
  - build -n ${param:jobs} -D EDK2_OUT_DIR=${param:builddir} ${param:join_space}
  postbuild: []
  artifacts:
    EDK2: ${param:builddir}/RELEASE_GCC5/FV/FVP_AARCH64_EFI.fd
edk2flash:
  repo: {}
  sourcedir: null
  builddir: null
  toolchain: null
  params: {}
  prebuild: []
  build: []
  postbuild: []
  artifacts:
    EDK2FLASH: ${param:configdir}/edk2-flash.img
tfa:
  repo:
    .:
      remote: https://git.trustedfirmware.org/TF-A/trusted-firmware-a.git
      revision: v2.8.0
  sourcedir: null
  builddir: null
  toolchain: aarch64-none-elf-
  params:
    PLAT: fvp
    DEBUG: 0
    LOG_LEVEL: 40
    ARM_DISABLE_TRUSTED_WDOG: 1
    FVP_HW_CONFIG_DTS: fdts/fvp-base-gicv3-psci-1t.dts
    BL33: ${artifact:EDK2}
    ARM_ARCH_MINOR: 5
    ENABLE_SVE_FOR_NS: 1
    ENABLE_SVE_FOR_SWD: 1
    CTX_INCLUDE_PAUTH_REGS: 1
    BRANCH_PROTECTION: 1
    CTX_INCLUDE_MTE_REGS: 1
    ENABLE_FEAT_HCX: 1
    CTX_INCLUDE_AARCH32_REGS: 0
    ENABLE_SME_FOR_NS: 1
    ENABLE_SME_FOR_SWD: 1
  prebuild: []
```

(continues on next page)

(continued from previous page)

```
    build:
    - make BUILD_BASE=${param:builddir} ${param:join_equal} all fip
    postbuild: []
    artifacts:
      BL1: ${param:builddir}/fvp/release/bl1.bin
      BL2: ${param:builddir}/fvp/release/bl2.bin
      BL31: ${param:builddir}/fvp/release/bl31.bin
      FIP: ${param:builddir}/fvp/release/fip.bin
artifacts: {}
run:
  name: FVP_Base_RevC-2xAEMvA
  rtvars:
    LOCAL_NET_PORT:
      type: string
      value: 8022
    BL1:
      type: path
      value: ${artifact:BL1}
    FIP:
      type: path
      value: ${artifact:FIP}
    DTB:
      type: path
      value: ${artifact:DTB}
    CMDLINE:
      type: string
      value: console=ttyAMA0 earlycon=pl011,0x1c090000 root=/dev/vda ip=dhcp
    KERNEL:
      type: path
      value: null
    ROOTFS:
      type: path
      value: ''
    EDK2FLASH:
      type: path
      value: ${artifact:EDK2FLASH}
  params:
    -C bp.dram_size: 4
    -C cluster0.NUM_CORES: 4
    -C cluster1.NUM_CORES: 4
    -C cluster0.PA_SIZE: 48
    -C cluster1.PA_SIZE: 48
    --stat: null
    -C bp.vis.disable_visualisation: 1
    -C bp.dram_metadata.is_enabled: 1
    -C bp.refcounter.non_arch_start_at_default: 1
    -C bp.refcounter.use_real_time: 0
    -C bp.secure_memory: 1
    -C bp.ve_sysregs.exit_on_shutdown: 1
    -C pctl.startup: 0.0.0.0
    -C cluster0.clear_reg_top_eret: 2
    -C cluster1.clear_reg_top_eret: 2
```

(continues on next page)

(continued from previous page)

```
    -C bp.smsc_91c111.enabled: 1
    -C bp.hostbridge.userNetworking: 1
    -C bp.hostbridge.userNetPorts: ${rtvar:LOCAL_NET_PORT}=22
    -C cache_state_modelled: 0
    -C cluster0.stage12_tlb_size: 1024
    -C cluster1.stage12_tlb_size: 1024
    -C cluster0.check_memory_attributes: 0
    -C cluster1.check_memory_attributes: 0
    -C cluster0.gicv3.cpuintf-mmap-access-level: 2
    -C cluster1.gicv3.cpuintf-mmap-access-level: 2
    -C cluster0.gicv3.without-DS-support: 1
    -C cluster1.gicv3.without-DS-support: 1
    -C cluster0.gicv4.mask-virtual-interrupt: 1
    -C cluster1.gicv4.mask-virtual-interrupt: 1
    -C pci.pci_smmuv3.mmu.SMMU_AIDR: 2
    -C pci.pci_smmuv3.mmu.SMMU_IDR0: 4592187
    -C pci.pci_smmuv3.mmu.SMMU_IDR1: 6291458
    -C pci.pci_smmuv3.mmu.SMMU_IDR3: 5908
    -C pci.pci_smmuv3.mmu.SMMU_IDR5: 4294902901
    -C pci.pci_smmuv3.mmu.SMMU_S_IDR1: 2684354562
    -C pci.pci_smmuv3.mmu.SMMU_S_IDR2: 0
    -C pci.pci_smmuv3.mmu.SMMU_S_IDR3: 0
    -C bp.virtio_rng.enabled: 1
    -C bp.secureflashloader.fname: ${rtvar:BL1}
    -C bp.flashloader0.fname: ${rtvar:FIP}
    -C bp.virtioblockdevice.image_path: ${rtvar:ROOTFS}
    -C cluster0.cpu0.semihosting-cwd: $${SEMIHOSTDIR}
    -C bp.flashloader1.fname: ${rtvar:EDK2FLASH}
    -C cluster0.has_16k_granule: 1
    -C cluster1.has_16k_granule: 1
    -C cluster0.has_arm_v8-1: 1
    -C cluster1.has_arm_v8-1: 1
    -C cluster0.has_large_system_ext: 1
    -C cluster1.has_large_system_ext: 1
    -C cluster0.has_arm_v8-2: 1
    -C cluster1.has_arm_v8-2: 1
    -C cluster0.has_large_va: 1
    -C cluster1.has_large_va: 1
    --plugin: $$(which ScalableVectorExtension.so)
    -C cluster0.has_arm_v8-3: 1
    -C cluster1.has_arm_v8-3: 1
    -C cluster0.has_arm_v8-4: 1
    -C cluster1.has_arm_v8-4: 1
    -C cluster0.has_amu: 1
    -C cluster1.has_amu: 1
    -C cluster0.has_arm_v8-5: 1
    -C cluster1.has_arm_v8-5: 1
    -C cluster0.has_branch_target_exception: 1
    -C cluster1.has_branch_target_exception: 1
    -C cluster0.has_rndr: 1
    -C cluster1.has_rndr: 1
    -C cluster0.memory_tagging_support_level: 3
```

(continues on next page)

(continued from previous page)

```
    -C cluster1.memory_tagging_support_level: 3
    -C cluster0.has_arm_v8-6: 1
    -C cluster1.has_arm_v8-6: 1
    -C cluster0.ecv_support_level: 2
    -C cluster1.ecv_support_level: 2
    -C cluster0.enhanced_pac2_level: 3
    -C cluster1.enhanced_pac2_level: 3
    -C cluster0.has_arm_v8-7: 1
    -C cluster1.has_arm_v8-7: 1
    -C cluster0.has_arm_v8-8: 1
    -C cluster1.has_arm_v8-8: 1
    -C cluster0.has_const_pac: 1
    -C cluster1.has_const_pac: 1
    -C cluster0.has_hpmn0: 1
    -C cluster1.has_hpmn0: 1
    -C cluster0.pmb_idr_external_abort: 1
    -C cluster1.pmb_idr_external_abort: 1
    -C cluster0.has_arm_v9-0: 1
    -C cluster1.has_arm_v9-0: 1
    -C cluster0.max_32bit_el: 0
    -C cluster1.max_32bit_el: 0
    -C SVE.ScalableVectorExtension.has_sve2: 1
    -C cluster0.has_arm_v9-1: 1
    -C cluster1.has_arm_v9-1: 1
    -C cluster0.has_arm_v9-2: 1
    -C cluster1.has_arm_v9-2: 1
    -C cluster0.has_brbe: 1
    -C cluster1.has_brbe: 1
    -C SVE.ScalableVectorExtension.has_sme: 1
    -C cluster0.has_arm_v9-3: 1
    -C cluster1.has_arm_v9-3: 1
    -C cluster0.has_brbe_v1p1: 1
    -C cluster1.has_brbe_v1p1: 1
  prerun:
  - SEMIHOSTDIR=`mktemp -d`
  - function finish { rm -rf $$SEMIHOSTDIR; }
  - trap finish EXIT
  - cp ${rtvar:KERNEL} $${SEMIHOSTDIR}/Image
  - cp ${rtvar:DTB} $${SEMIHOSTDIR}/fdt.dtb
  - cat <<EOF > $${SEMIHOSTDIR}/startup.nsh
  - Image dtb=fdt.dtb ${rtvar:CMDLINE}
  - EOF
  run: []
  terminals:
    bp.terminal_0:
      friendly: ''
      port_regex: 'terminal_0: Listening for serial connection on port (\d+)'
      type: stdinout
      no_color: true
      no_escapes: 'EFI stub: Booting Linux Kernel...'
    bp.terminal_1:
      friendly: edk2
```

(continues on next page)

(continued from previous page)

```
    port_regex: 'terminal_1: Listening for serial connection on port (\d+)'
    type: stdout
  bp.terminal_2:
    friendly: term2
    port_regex: 'terminal_2: Listening for serial connection on port (\d+)'
    type: stdout
  bp.terminal_3:
    friendly: term3
    port_regex: 'terminal_3: Listening for serial connection on port (\d+)'
    type: stdout
```

1.2.2 Run-Times

Shrinkwrap uses a "runtime" to execute all of its shell commands and allows the user to choose which runtime to use. Both the design and implementation of this is borrowed from Tuxmake.

Shrinkwrap supports the following set of runtimes:

runtime	description
null	Shell commands are executed natively on the user's system. The user is responsible for ensuring the the required toolchain, environment variables and any other dependencies are set up.
docker	(default). Shell commands are executed in a docker container. By default, the official shrinkwrap image will be pulled and used, which contains all dependencies already setup.
docker-local	Like docker, but will only look for the container image on the local system. Will not attempt to pull over the network.
podman	Shell commands are executed in a podman container. By default, the official shrinkwrap image will be pulled and used, which contains all dependencies already setup.
podman-local	Like podman, but will only look for the container image on the local system. Will not attempt to pull over the network.

The desired runtime can be specified using the `--runtime` option, which is a top-level argument (must come before the command):

```
shrinkwrap --runtime=<name> ...
```

If using a container runtime (anything other than null), a custom image can optionally be specified. If omitted, the official shrinkwrap image is used:

```
shrinkwrap --runtime=<name> --image=<name> ...
```

Container Image Variants

Shrinkwrap runs on both x86_64 and aarch64 architectures, and provides multiarch container images so that the correct variant is automatically selected for your platform. Images are automatically downloaded by shrinkwrap when the `docker` or `podman` runtime is selected. Images are available on Docker Hub and can be freely downloaded without the need for an account.

image name	description
docker.io/shrinkwraptool/base-slim-nofvp:latest	Contains all toolchains and other dependencies required to build all standard configs. Can be used as a base to create an image with a custom FVP.
docker.io/shrinkwraptool/base-slim:latest	As per `shrinkwraptool/base-slim-nofvp:latest` but also contains the Base_RevC-2xAEMvA FVP. This is suffcient for most use cases and is much smaller than the `full` variant.
docker.io/shrinkwraptool/base-full-nofvp:latest	Builds upon `shrinkwraptool/base-slim:latest`, adding aarch32 toolchains (both arm-none-eabi and arm-linux-gnueabihf). These are not needed for standard configs, but will be required if creating a custom config that includes (e.g.) SCP FW. Separated out due to big size increase.
docker.io/shrinkwraptool/base-full:latest	As per `shrinkwraptool/base-full-nofvp:latest` but also contains the Base_RevC-2xAEMvA FVP.

Runtime Requirements

The best way to understand the requirements for the packages available within the runtime is to look at the dockerfiles for the official shrinkwrap images. These are available at `docker/Dockerfile.*`.

Build Container Image Locally

If you have a need to build the shrinkwrap container images on your local system, you can do it as follows:

```
cd docker
./build.sh local
```

This will build a set of images called:

- `shrinkwraptool/base-slim:local-<ARCH>`
- `shrinkwraptool/base-slim-nofvp:local-<ARCH>`
- `shrinkwraptool/base-full:local-<ARCH>`
- `shrinkwraptool/base-full-nofvp:local-<ARCH>`

To use a locally built image, call shrinkwrap as follows if running on an x86_64 system:

```
shrinkwrap --runtime=<name>-local --image=shrinkwraptool/base-slim:local-x86_64 ...
```

Or like this if running on an aarch64 system:

```
shrinkwrap --runtime=<name>-local --image=shrinkwraptool/base-slim:local-aarch64 ...
```

where <name> is either docker or podman. Note that because the image is not on Docker Hub, the <name>-local runtime is required to prevent Shrinkwrap from erroneously trying to download an update.

1.2.3 Commands

For documentation on the commands that shrinkwrap supports, add `--help` to the command line.

For top-level help:

```
shrinkwrap --help
```

For help on a specific command:

```
shrinkwrap inspect --help
shrinkwrap build --help
shrinkwrap buildall --help
shrinkwrap clean --help
shrinkwrap run --help
shrinkwrap process --help
```

Todo: Automate importing the help pages into this documentation.

1.2.4 Config Model

A config is a yaml file that defines everything about a given configuration. This includes:

- meta data (e.g. its name, description, etc)
- the components that should be built
- how those components are built
- dependencies between components
- what artifacts are produced
- how to configure the fvp
- how to load and run the artifacts on the fvp

A config is declarative; the user declares how things relate and how things should be done, and the tool extracts the information to decide exactly what should be done and in what order in order to complete a task. All the data is contained within the config and drives the tool. This way, shrinkwrap is highly extensible. The user specifies the config(s) that should be used when invoking shrinkwrap.

Merging Configs

A config is laid out as a hierarchical data structure, using nested dictionaries. This suits it very well to being split into partial configs that are merged together into a single, final config. This allows maximal reuse of the config fragments and improves maintainability. Each config can optionally define a set of foundational `layers` which it then builds upon. Furthermore, the user can optionally specify a custom `overlay` config on the command line. Layers are merged in order according to the following rules:

For each leaf key in the union of the hierarchical dictionaries:

- If the upper value is null or not present, then the lower value is taken
- If both the upper and lower values are lists, then the final value is the lower list with the upper list appended to its end.

- In all other cases the upper value is taken

You can use the `process` command to merge configs and see the resulting output to get a better feel for how it works. See *Commands*.

Merging Example

Listing 1: lower config

```
people:
  Iris:
    age: 2
    likes:
      - Peppa Pig
      - Bananas
```

Listing 2: upper config

```
people:
  Iris:
    age: 3
    likes:
      - Peas
  James:
    age: 6
    likes:
      - FIFA
```

Listing 3: merged result

```
people:
  Iris:
    age: 3
    likes:
      - Peppa Pig
      - Bananas
      - Peas
  James:
    age: 6
    likes:
      - FIFA
```

Macros

Macros are placeholders that can be specified in various parts of a config yaml file, which are substituted ("resolved") with information that is only known at build-time or run-time. There are specific rules about which macros can be used in which parts of the config, and about the order in which they get substituted.

Macros take the following form:

```
${<type>:[<name>]}
```

where:

- `type` is a required namespace for the macro family
- `name` is an optional name for the macro within its namespace. For some macro types, there are a fixed set of names. For others, the names are defined by the config itself.

You can use the `process` command to resolve macros and see the resulting output to get a better feel for how they work. See *Commands*.

Defined Macros

macro	scope	description
`${param:sourcedir}`	build.<component>.{params, prebuild, build, postbuild, artifacts}	Directory in which the component's source code is located.
`${param:builddir}`	build.<component>.{params, prebuild, build, postbuild, artifacts}	Directory in which the component should be built, if the component's build system supports separation of source and build trees.
`${param:configdir}`	build.<component>.{params, prebuild, build, postbuild, artifacts}	Directory containing the config store. This MUST only be used for resolving files that already exist in the store.
`${param:jobs}`	build.<component>.{params, prebuild, build, postbuild, artifacts}	Maximum number of low level parallel jobs specified on the command line. To be passed to (e.g.) make as `-j${param:jobs}`.
`${btvar:<name>}`	build.<component>.{sourcedir, builddir, repo, toolchain, params, prebuild, build, postbuild, artifacts}, run.rtvars	Build-time variables. The variable names, along with default values are declared in buildex.btvars, and the user may override the value on the command line.
`${param:joined:<name>}`	build.<component>.{prebuild, build, postbuild}	String containing all of the component's parameters (from its params dictionary), concatenated as `key=value` pairs.
`${param:joined_space:<name>}`	build.<component>.{prebuild, build, postbuild}	String containing all of the component's parameters (from its params dictionary), concatenated as `key value` pairs.
`${artifact:<name>}`	build.<component>.{params, prebuild, build, postbuild, artifacts}, build.btvars	Build path of an artifact declared by another component. Usage of these macros determine the component build dependency graph.
`${artifact:runtime:<name>}`	run.rtvars	Package path of an artifact.
`${rtvar:<name>}`	run.params	Run-time variables. The variable names, along with default values are declared in run.rtvars, and the user may override the value on the command line.

Schema

Top-Level keys

The following is the set of top-level public keys that should be defined by a config. There are some additional private keys that the tool will add (and make visible as part of the `process` command), but these are subject to change and not documented.

key	type	description
description	string	A human-readable description of what the config contains and does. Displayed by the `inspect` command.
concrete	boolean	true if the config is intended to be directly built and run, or false if it is intended as a fragment to be included in other configs.
build	dictionary	Contains all the components to be built. The key is the component name and the value is a dictionary.
run	dictionary	Contains all the information about how to run the built artifacts on the FVP.

build section

The build section, contains a dictionary of components that must be built. The keys are the component names and the values are themselves dictionaries, each containing the component meta data.

buildex section

When the schema was originally created, we made a mistake. The components should have been under `build: components:`, allowing room for new build data to be added under `build:` without being confused for components. In order to retrofit a solution without breaking compatibility, the buildex section is created.

key	type	description
btvars	dictionary	Build-Time variables. Keys are the variable names and values are a dictionary with keys 'type' (which must be one of 'path' and 'string') and 'value' (which takes the default value). Build-Time variables can be overridden by the user at the command line.

component section

key	type	description
repo	dictionary	Specifies information about the git repo(s) that must be cloned and checked out. By default, Shrinkwrap syncs the git repo to the specified revision when building. `--no-sync` or `--no-sync-all` can be used to tell Shrinkwrap to build it in whatever state the user left it in. Not required if `sourcedir` is provided.
sourcedir	string	If specified, points to the path on disk where the source repo can be found. Useful for developer use cases where a local repo already exists.
builddir	string	If specified, the location where the component will be built. If not specified, shrinkwrap allocates its own location based on SHRINKWRAP_BUILD.
toolchain	string	Defines the toolchain to be used for compilation. Value is set as CROSS_COMPILE environment variable before invoking any prebuild/build/postbuild commands. When using the standard image with a container runtime, the options are: `aarch64-none-elf-`, `arm-none-eabi-`, `aarch64-linux-gnu-`, or `arm-linux-gnueabihf-`.
stderrfilt	bool	Optional, defaults to false. When true, and –verbose is not specified, filters stderr of the component's build task so that only lines containing 'error' and 'warning' are output. Everything else is suppressed. Useful for EDK2 which is extremely chatty.
params	dictionary	Optional set of key:value pairs. When building most components, they require a set of parameters to be passed. By setting them out as a dictionary, it is easy to override and add to them in higher layers. See `${param:join_*}` macros.
prebuild	list	List of shell commands to be executed during component build before the `build` list.
build	list	List of shell commands to be executed during component build.
postbuild	list	List of shell commands to be executed during component build after the `build` list.
artifacts	dictionary	Set of artifacts (files and/or directories) that the component exports. Key is artifact name and value is path to built artifact. Other components can reference them with the `${artifact:<name>}` macros. Used to determine build dependencies.

run section

key	type	description
name	string	Name of the FVP binary, which must be in $PATH.
rtvars	dictionary	Run-Time variables. Keys are the variable names and values are a dictionary with keys 'type' (which must be one of 'path' and 'string') and 'value' (which takes the default value). Run-Time variables can be overridden by the user at the command line.
params	dictionary	Dictionary of parameters to be passed to the FVP. Similar to the component's params, laying these out in a dictionary makes it easy for higher layers to override and add parameters.
prerun	list	List of shell commands to be executed before the FVP is started.
terminals	dictionary	Describes the set of UART terminals available for the FVP. key is the terminal parameter name known to the FVP (e.g. `bp.terminal_0`) See below for format of the value.

terminal section

key	type	description
friendly	string	Label to display against the terminal when muxing to stdout. An empty string disables the prefix for the output.
port_regex	string	Regex to use to find the TCP port of the terminal when parsing the FVP stdout. Must have single capture group.
type	enum-string	Terminal type. See below for options.
no_color	boolean	Optional (defaults to false, only applies to ['stdout', 'stdinout'] types): If true, output from this terminal is not color-coded. If this terminal carries the interactive shell, it is advised to set this to true to prevent interferring with the shell's escape sequences. –no-color command line option causes this to behave as if set to true.
no_escapes	bool/string	Optional (defaults to false, only applies to ['stdout', 'stdinout'] types): If true, strips any escape sequences from the output stream before forwarding to the terminal. If a string, behaves as if true until the string is found in the output, which sets it to false. Useful to expunge escape sequences from EDK2 during boot.
log-file	string	Optional (defaults to none, only applies to ['stdout', 'stdinout'] types): Specifies path to a log file where all output to the terminal will be duplicated.

Terminal types:

- **stdout**: Mux output to stdout. Do not supply any input.
- **stdinout**: Mux output to stdout. Forward stdin to its input. Max of 1 of these types allowed.
- **telnet**: Shrinkwrap will print out a telnet command to run in a separate terminal to get a unique interactive terminal.
- **xterm**: Shrinkwrap will automatically launch xterm to provide a unique interactive terminal. Only works when runtime=null.

1.2.5 Config Store

Shrinkwrap ships with the following standard set of configs for use out-of-the-box:

bootwrapper.yaml

Description

Best choice for: I have a linux-system.axf boot-wrapper and want to run it.

This config does not build any components (although shrinkwrap still requires you to build it before running). Instead the user is expected to provide a boot-wrapper executable (usually called linux-system.axf) as the BOOTWRAPPER rtvar, which will be executed in the FVP. A ROOTFS can be optionally provided. If present it is loaded into the virtio block device (/dev/vda).

Concrete

True

Build-Time Variables

btvar	default

Run-Time Variables

rtvar	default
LOCAL_NET_PORT	8022
BOOTWRAPPER	<null>
ROOTFS	<empty>

buildroot.yaml

Description

Generates a very simple rootfs as an ext2/4 image. Higher layers can modify the buildroot config by adding commands to prebuild.

Concrete

True

Build-Time Variables

btvar	default

Run-Time Variables

rtvar	default

cca-3world.yaml

Description

Brings together a software stack to demonstrate Arm CCA running on FVP in a three-world configuration. Includes TF-A in root world, RMM in realm world, and EDK2 and Linux in Normal world on the host. Guests can be launched in-realm in a number of configurations using kvmtool. EDK2 can be optionally used as guest FW.

If the user provides an ext2/4 filesystem image via the GUEST_ROOTFS btvar, a guest disk image is created that includes a FAT16 partition containing the guest kernel (to be loaded by the guest EDK2 FW), and the provided filesystem as the rootfs. The user can provide their own filesystem image, or alternatively use a simple buildroot image created with buildroot.yaml:

```
$ shrinkwrap build cca-3world.yaml --overlay buildroot.yaml --btvar GUEST_ROOTFS='$
↪{artifact:BUILDROOT}'
```

Once built, the user must get some of the generated artifacts into the FVP environment. This can either be done by copying them to the host's rootfs or by sharing them into the FVP using 9p.

If copying to the rootfs, something like this should work. For simplicity, this example reuses the guest filesystem generated with buildroot as the host's rootfs, after resizing it so that there is room for the guest's rootfs:

```
$ cd ~/.shrinkwrap/package/cca-3world
$ e2fsck -fp rootfs.ext2
$ resize2fs rootfs.ext2 256M
$ sudo su
# mkdir mnt
# mount rootfs.ext2 mnt
# mkdir mnt/cca
# cp guest-disk.img KVMTOOL_EFI.fd lkvm mnt/cca/.
# umount mnt
# rm -rf mnt
# exit
```

Now you can boot the host, using the rootfs we just modified, either using DT:

```
$ shrinkwrap run cca-3world.yaml --rtvar ROOTFS=rootfs.ext2
```

Or alternatively, using ACPI:

```
$ shrinkwrap run cca-3world.yaml -r ROOTFS=rootfs.ext2 --rtvar CMDLINE="mem=1G earlycon␣
↪root=/dev/vda ip=dhcp acpi=force"
```

Or if taking the shared directory approach, simply boot the host with the SHARE rtvar. This only works for DT-based environments:

```
$ cd ~/.shrinkwrap/package/cca-3world
$ shrinkwrap run cca-3world.yaml --rtvar ROOTFS=rootfs.ext2 --rtvar SHARE=.
```

Finally, once the host has booted, log in as "root" (no password), and launch a realm using kvmtool. Note the mount command is only required if sharing a directory:

```
# mkdir /cca
# mount -t 9p -o trans=virtio,version=9p2000.L FM /cca
```

(continues on next page)

(continued from previous page)

```
# cd /cca
# ./lkvm run --realm --disable-sve --irqchip=gicv3-its --firmware KVMTOOL_EFI.fd -c 1 -m␣
↪512 --no-pvtime --force-pci --disk guest-disk.img --measurement-algo=sha256
```

Be patient while this boots to the UEFI shell. Navigate to "Boot Manager", then "UEFI Shell" and wait for the startup.nsh script to execute, which will launch the kernel. Continue to be patient, and eventually you will land at a login prompt. Login as "root" (no password).

This config also builds kvm-unit-tests, which can be run in the realm instead of Linux. It is also possible to launch Linux without using EDK2 as the guest FW.

Concrete

True

Build-Time Variables

btvar	default
GUEST_ROOTFS	<empty>

Run-Time Variables

rtvar	default
LOCAL_NET_PORT	8022
BL1	${artifact:BL1}
FIP	${artifact:FIP}
DTB	${artifact:DTB}
CMDLINE	console=ttyAMA0 earlycon=pl011,0x1c090000 root=/dev/vda ip=dhcp
KERNEL	${artifact:KERNEL}
ROOTFS	<empty>
SHARE	<empty>
EDK2FLASH	${artifact:EDK2FLASH}

cca-4world.yaml

Description

Builds on cca-3world.yaml, and adds support for running Hafnium along with some secure partitions in Secure World. Build with:

```
$ shrinkwrap --image shrinkwraptool/base-full build cca-4world.yaml --overlay buildroot.
↪yaml --btvar GUEST_ROOTFS='${artifact:BUILDROOT}'
```

Then run the model with:

```
$ cd ~/.shrinkwrap/package/cca-4world
$ shrinkwrap run cca-4world.yaml --rtvar ROOTFS=rootfs.ext2 --rtvar SHARE=.
```

Once the host has booted, log in as “root” (no password).

Secure partitions can be enumerated by:

```
# cat /sys/devices/arm-ffa-*/uuid
b4b5671e-4a90-4fe1-b81f-fb13dae1dacb
d1582309-f023-47b9-827c-4464f5578fc8
79b55c73-1d8c-44b9-8593-61e1770ad8d2
eaba83d8-baaf-4eaf-8144-f7fdcbe544a7
```

See cca-3worlds.yaml config *Description* if willing to launch a realm using kvmtool.

Concrete

True

Build-Time Variables

btvar	default
GUEST_ROOTFS	<empty>

Run-Time Variables

rtvar	default
LOCAL_NET_PORT	8022
BL1	${artifact:BL1}
FIP	${artifact:FIP}
DTB	${artifact:DTB}
CMDLINE	console=ttyAMA0 earlycon=pl011,0x1c090000 root=/dev/vda ip=dhcp
KERNEL	${artifact:KERNEL}
ROOTFS	<empty>
SHARE	<empty>
EDK2FLASH	${artifact:EDK2FLASH}

ffa-tftf.yaml

Description

Brings together a software stack to demonstrate Arm FF-A running on FVP. Includes TF-A in secure EL3, Hafnium in secure EL2 and some demo TF-A test secure partitions.

Concrete

True

Build-Time Variables

btvar	default

Run-Time Variables

rtvar	default
LOCAL_NET_PORT	8022
BL1	${artifact:BL1}
FIP	${artifact:FIP}
DTB	${artifact:DTB}
CMDLINE	console=ttyAMA0 earlycon=pl011,0x1c090000 root=/dev/vda ip=dhcp
KERNEL	<null>
ROOTFS	<empty>
SHARE	<empty>
EDK2FLASH	${artifact:EDK2FLASH}

ns-edk2.yaml

Description

Best choice for: I want to run Linux on FVP, booting with ACPI/DT, and have easy control over its command line.

Brings together TF-A and EDK2 to provide a simple non-secure world environment running on FVP. Allows easy specification of the kernel image and command line, and rootfs at runtime (see rtvars). ACPI is provided by UEFI.

An extra rtvar is added (DTB) which allows specification of a custom device tree. By default (if not overriding the rtvar), the upstream kernel device tree is used. DT is enabled by default. Use 'acpi=force' to enable ACPI boot.

By default (if not overriding the rtvars) a sensible command line is used that will set up the console for logging and attempt to mount the rootfs image from the FVP's virtio block device. However the default rootfs image is empty, so the kernel will panic when attempting to mount; the user must supply a rootfs if it is required that the kernel completes its boot. No default kernel image is supplied and the config will refuse to run unless it is explicitly specified.

Note that by default, a pre-canned flash image is loaded into the model, which contains UEFI variables directing EDK2 to boot to the shell. This will cause startup.nsh to be executed and will start the kernel boot. This way everything is automatic. By default, all EDK2 output is muxed to stdout. If you prefer booting UEFI to its UI, override the EDK2FLASH rtvar with an empty string and override terminals.'bp.terminal_0'.type to 'telnet'.

When booting with device tree, a directory can optionally be shared from the host system into the Linux environment running in the FVP. To do so, set the SHARE rtvar to the desired directory, then mount the share inside the FVP with the following (or automate it in fstab):

```
# mkdir /share
# mount -t 9p -o trans=virtio,version=9p2000.L FM /share
```

Concrete

True

Build-Time Variables

btvar	default

Run-Time Variables

rtvar	default
LOCAL_NET_PORT	8022
BL1	${artifact:BL1}
FIP	${artifact:FIP}
DTB	${artifact:DTB}
CMDLINE	console=ttyAMA0 earlycon=pl011,0x1c090000 root=/dev/vda ip=dhcp
KERNEL	<null>
ROOTFS	<empty>
SHARE	<empty>
EDK2FLASH	${artifact:EDK2FLASH}

ns-preload.yaml

Description

Best choice for: I just want to run Linux on FVP.

A simple, non-secure-only configuration where all components are preloaded into memory (TF-A's BL31, DTB and kernel). The system resets directly to BL31. Allows easy specification of a custom command line at build-time (via build.dt.params dictionary) and specification of the device tree, kernel image and rootfs at run-time (see rtvars).

By default (if not overriding the rtvars), the upstream kernel device tree is used along with a sensible command line that will set up the console for logging and attempt to mount the rootfs image from the FVP's virtio block device. However the default rootfs image is empty, so the kernel will panic when attempting to mount; the user must supply a rootfs if it is required that the kernel completes its boot. No default kernel image is supplied and the config will refuse to run unless it is explicitly specified. Note: If specifying a custom dtb at runtime, this will also override any command line specified at build time, since the command line is added to the chosen node of the default dtb.

A directory can optionally be shared from the host system into the Linux environment running in the FVP. To do so, set the SHARE rtvar to the desired directory, then mount the share inside the FVP with the following (or automate it in fstab):

```
# mkdir /share
# mount -t 9p -o trans=virtio,version=9p2000.L FM /share
```

Concrete

True

Build-Time Variables

btvar	default

Run-Time Variables

rtvar	default
LOCAL_NET_PORT	8022
BL31	${artifact:BL31}
DTB	${artifact:DTB}
KERNEL	<null>
ROOTFS	<empty>
SHARE	<empty>

1.2.6 Shrinkwrap Recipes

This page contains an unordered list of Shrinkwrap usage examples intended to demonstrate how Shrinkwrap can solve various problems.

Override Component Version and/or Location

You can change many, many configuration options by overlaying a config on top of an existing config. Here we modify the revision and remote repository of the TF-A component from its default (defined in tfa-base.yaml). You could also specify the revision as a SHA or branch.

Create a file called `my-overlay.yaml`:

```
build:
  tfa:
    repo:
      remote: https://github.com/ARM-software/arm-trusted-firmware.git
      revision: v2.9
```

Optionally, you can view the final, merged config as follows:

```
shrinkwrap process --action=merge --overlay=my-overlay.yaml ns-preload.yaml
```

Now do a build, passing in the overlay:

```
shrinkwrap build --overlay=my-overlay.yaml ns-preload.yaml
```

Finally, boot the config:

```
shrinkwrap run --rtvar=KERNEL=path/to/Image ns-preload.yaml
```

Reuse Existing Local Repo for Component

By default, shrinkwrap will sync the git repos for all required components to a private location (<SHRINKWRAP_BUILD>/source/<config_name>/<component_name>) the first time you build a given config. However, sometimes you want shrinkwrap to reuse a repo that already exists on your local system. In this case, Shrinkwrap will build this source into its own private build tree, leaving the source tree unmodified.

Warning: Components support building in a tree separate from the source to differing degrees. For example, TF-A will always build fiptool in the source tree, although it will build all the FW components in the correct build tree. So depending on the component you are sharing source for, you may see some build artifacts appear.

Create a file called `my-overlay.yaml`:

```
build:
  tfa:
    sourcedir: /path/to/my/tfa/git/repo
```

Now do a build, passing in the overlay:

```
shrinkwrap build --overlay=my-overlay.yaml ns-preload.yaml
```

Changing Between Arm Architecture Revisions

Shrinkwrap comes with a set of configs that can be overlaid onto the primary config in order to modify the targeted Arm architecture revision. These overlays provide all the required modifications for the TF-A build configuration and the FVP run configuration. Each architecture revision includes all mandatory features associated with that extension as well as a selection of sensible/common optional features.

`Armv8.0` - `Armv8.8` and `Armv9.0` - `Armv9.3` are currently supported. The yaml files are in the `arch` subdirectory of the config store. (You can see them by running the `inspect` command with the `--all` option).

The below will build the `ns-edk2` config for Armv8.8 and run it on the FVP configured for the same revision.

```
shrinkwrap build ns-edk2.yaml --overlay=arch/v8.8.yaml
shrinkwrap run ns-edk2.yaml --rtvar=KERNEL=path/to/Image
```

Warning: Some components (notably TF-A) fail to incrementally build when changing their make parameters. Therefore, if you want to change the architecture revision for a config that has already been built, you must first clean tfa. See *Workaround for TF-A not Noticing Modified Build Params*.

Explicitly Clean a Config or Component

If the `build` command is invoked multiple times, Shrinkwrap will always attempt to do an incremental build. This enables a developer to modify the source and easily rebuild and run the result. However, sometimes it is useful to explicitly clean a component (or all the components within a config) to force it to be rebuilt from scratch. Shrinkwrap includes a `clean` command for this.

Clean an entire config (all components in config):

```
shrinkwrap clean ns-edk2.yaml
```

Clean a specific set of components from a config (in this case, clean the tfa and dt components):

```
shrinkwrap clean ns-edk2.yaml --filter=tfa --filter=dt
```

Then rebuild the config and the cleaned components are rebuilt from scratch:

```
shrinkwrap build ns-edk2.yaml
```

Workaround for TF-A not Noticing Modified Build Params

TF-A is not good at noticing when its build parameters change. If you have already built TF-A, then attempt to do an incremental build with different parameters, you rarely get what you expect. This happens a lot when using the arch/vX.Y.yaml overlays, because different architecture revisions need to specify different TF-A build parameters.

Work around this by explicitly cleaning TF-A when changing architecture revisions:

```
shrinkwrap build ns-edk2.yaml --overlay=arch/v8.7.yaml
shrinkwrap clean ns-edk2.yaml --filter=tfa
shrinkwrap build ns-edk2.yaml --overlay=arch/v9.3.yaml
```

Use a Custom FVP Version

By default, the `run` command will use the FVP that is bundled with the latest published shrinkwrap docker image. Sometimes you might want to use a different version though. In this case, the simplest approach is to install the FVP on your system, ensuring that the required directories are in your PATH, and invoke `shrinkwrap run` with the `null` runtime.

Shrinkwrap expects both the FVP binary (e.g. FVP_Base_RevC-2xAEMvA) and its plugins (e.g. ScalableVectorExtension.so) to be on your path. The example below shows downloading and untaring the FVP and adding the required directories to the PATH.

```
wget -q -O FVP_Base_RevC-2xAEMvA_11.18_16_Linux64.tgz https://developer.arm.com/-/media/
↪Files/downloads/ecosystem-models/FVP_Base_RevC-2xAEMvA_11.18_16_Linux64.tgz
tar xf FVP_Base_RevC-2xAEMvA_11.18_16_Linux64.tgz
export PATH=$PWD/Base_RevC_AEMvA_pkg/models/Linux64_GCC-9.3:$PWD/Base_RevC_AEMvA_pkg/
↪plugins/Linux64_GCC-9.3:$PATH
shrinkwrap build ns-edk2.yaml
shrinkwrap --runtime=null run ns-edk2.yaml --rtvar=KERNEL=path/to/Image
```

Use an Alternative Device Tree

There are a couple of ways to use an alternative device tree:

All provided concrete configs that use a device tree, expose a DTB rtvar with a default value. Users can override this value to provide an externally compiled DTB at **run-time**.

Alternatively, the dt-base.yaml config fragment can be passed a parameter at **build-time** that tells it to compile an alternative DTS file. dt-base.yaml builds the device tree for the FVP_Base_RevC-2xAEMvA FVP by default and is used by all the standard concrete configs that require a device tree and is available for use in defining custom configs.

Add the following to a higher layer of the config:

```
build:
  dt:
    prebuild:
      - DTS=foundation-v8-gicv3-psci.dts
```

Note that dt-base.yaml only accepts names of dts files that already exist in the device tree repo.

Accessing the FVP over Network when using Docker/Podman

When using the docker or podman runtimes, the FVP runs inside a container. This has a different IP address to the host system. Shrinkwrap helpfully prints out the runtime environment's IP address when starting the FVP. This is the IP address you need to use to (e.g.) connect the debugger or to SSH into the hosted Linux system.

Boot Linux with ACPI

`ns-edk2.yaml` uses EDK2 to boot Linux, and defaults to using the Device Tree. You can change the behaviour to boot with ACPI by passing `acpi=force` on the comand line:

```
shrinkwrap run ns-edk2.yaml --rtvar=KERNEL=path/to/Image --rtvar=CMDLINE=
↪"console=ttyAMA0 earlycon=pl011,0x1c090000 root=/dev/vda ip=dhcp acpi=force"
```

Pass Overlay Directly on Command Line

All of the previous examples that utilize overlays, put the overlay in a yaml file and pass the file name on the command line. Here is an example that runs the FVP as normal but saves all output from UART0 to uart0.log:

Create a file called `my-overlay.yaml`:

```
run:
  params:
    -C bp.pl011_uart0.out_file: uart0.log
```

Now run the FVP, passing in the overlay:

```
shrinkwrap run ns-edk2.yaml --rtvar=KERNEL=path/to/Image --overlay=my-overlay.yaml
```

However, it is also possible to pass an overlay, encoded as json, directly on the command line, without the need for a file. This is useful when the overlay is small. JSON allows the entire content to be encoded on a single line without having to care about whitespace, so is suited to this purpose.

This is equivalent:

```
shrinkwrap run ns-edk2.yaml --rtvar=KERNEL=path/to/Image --overlay='{"run":{"params":{"-
↪C bp.pl011_uart0.out_file":"uart0.log"}}}'
```

Provide SSH keys

The GIT subprocess used by Shrinkwrap to synchronise source code may require access to SSH keys. This is supported, via `ssh-agent`.

If `ssh-agent` is already running, it will be automatically detected (via the `SSH_AUTH_SOCK` environment variable) and used by Shrinkwrap.

Alternatively, the `--ssh-agent` option can be used to request Shrinkwrap to start an `ssh-agent` subprocess and add default keys.

```
shrinkwrap --ssh-agent build ...
```

In order to add only specified keys, the `--ssh-agent-key` can be used.

```
shrinkwrap \
  --ssh-agent-key ~/.ssh/my-first-key \
  --ssh-agent-key ~/.ssh/my-second-key \
  build ...
```

1.3 Developer Guide

1.3.1 Compile Documentation Locally

To build the docs locally, the following packages need to be installed on the host:

```
sudo apt-get install python3-pip
pip3 install -U -r documentation/requirements.txt
```

To build and generate the documentation in html format, run:

```
sphinx-build -b html -a -W documentation public
```

To render and explore the documentation, simply open *public/index.html* in a web browser.

1.4 License

The software is provided under the MIT license (below).

```
Copyright (c) 2022 Arm Limited

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to
deal in the Software without restriction, including without limitation the
rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the Software is
```

(continues on next page)

(continued from previous page)

```
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice (including the next
paragraph) shall be included in all copies or substantial portions of the
Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE SOFTWARE.
```

1.4.1 SPDX Identifiers

Individual files contain the following tag instead of the full license text.

```
SPDX-License-Identifier: MIT
```

This enables machine processing of license information based on the SPDX License Identifiers that are here available: http://spdx.org/licenses/