

Firmware project with Meson build system

Community Over Code Glasgow 2026

Karel Kočí

13.10.2026

The Point

POSIX gives ability to run same programs on NuttX and Linux.

Advantages:

- Options when selecting libraries
- Easier and faster testing (including CI/CD)
- Possibility to test with tools such as Valgrind

Workflow:

- Develop on PC including libraries
- Unit testing on PC
- Sw/sw integration on PC
- Sw/hw integration on real hardware

We need a good way to build project for both targets (for NuttX flat build).

NuttX external

Standard NuttX



```
cd nuttx  
./tools/configure.sh rv-virt:nsh  
make
```

External Applications and Libraries



configure-nuttx.sh

```
ln -sf ../project-libs nuttx/external
ln -sf ../project-apps apps/external
```

Apps Directory

project-apps/Make.defs

```
include $(wildcard $(APPDIR)/external/*/Make.defs)
```

project-apps/Makefile

```
MENUESC = "Project Applications"  
include $(APPDIR)/Directory.mk
```

A Project App

project-apps/project_hello/Make.defs

```
CONFIGURED_APPS += $(APPDIR)/external/project_hello
```

project-apps/project_hello/Makefile

```
include $(APPDIR)/Make.defs
PROGNAME  = $(CONFIG_NUTTX_USBNSH_PROGNAME)
PRIORITY  = $(CONFIG_NUTTX_USBNSH_PRIORITY)
STACKSIZE = $(CONFIG_NUTTX_USBNSH_STACKSIZE)
MODULE     = $(CONFIG_NUTTX_USBNSH)
MAINSRC   = project_hello_main.c
include $(APPDIR)/Application.mk
```

project-apps/project_hello/Kconfig ...

A Project App: In More Depth

```
all:: .built
context::
register::
    $(call REGISTER,$(PROGNAME_@),$(PRIORITY_@),$(STACKSIZE_@),$_main)

.built: $(AROBJJS)
    $(call ARLOCK, $(call CONVERT_PATH,$(BIN)), $^)

CFLAGS += ${DEFINE_PREFIX}main=$(addsuffix _main,$(PROGSYM_@))
```

External Board

```
mkdir board
```

```
cp -r nuttx/boards/risc-v/qemu-rv/rv-virt/{src,include,scripts} board/
```

board/defconfig (replace CONFIG_ARCH_BOARD*)

```
...
```

```
CONFIG_ARCH_BOARD_CUSTOM=y
```

```
CONFIG_ARCH_BOARD_CUSTOM_DIR="../board"
```

```
CONFIG_ARCH_BOARD_CUSTOM_DIR_RELPATH=y
```

```
CONFIG_ARCH_BOARD_CUSTOM_NAME="TEMPLATE"
```

configure-nuttx.sh

```
...
```

```
cp -f board/defconfig nuttx/.config
```

```
make -C nuttx olddefconfig
```

Bonus Round: Config verification

```
while read -r line; do
    [[ "$cnf" == CONFIG_* ]] || continue
    if [[ "${line#*=}" == "n" ]]; then
        if grep -q "^${line%*=}(y|m)$" nuttx/.config; then
            echo "Not correctly set: $line" >&2; exit 1
        fi
    else
        if ! grep -qF "$line" nuttx/.config; then
            echo "Not correctly set: $line" >&2; exit 1
        fi
    fi
done <"board/defconfig"
done
```

Meson



Requirements for Build System

- Build libraries and applications
- No special rules in build system for NuttX
- Support for external projects with minimal NuttX boilerplate

Quick Overview

Leading build system for building libraries and executables for Linux.

meson.build

```
project('tutorial', 'c')
libfoo = library('foo', files('foo.c'),
                include_directories=include_directories('.'))
executable('demo', files('main.c'), dependencies: [libfoo],
           install: true)
```

Outputs ninja or other build systems.

Compare with make and cmake

Make

- 😊 Generic build system (language)
- 😞 Can be tweaked to build almost anything
- 😞 Good build systems are complex
- 😞 Build targets standardization old and obsolete
- 🙄 No standard way to set toolchain
- 😞 No standard way to manage build time dependencies

CMake

- 😊 Generic build system (language)
- 😞 Can be tweaked to build almost anything
- 😊 Has build target standardization
- 😞 Build target standardization is optional
- 😊 Toolchain.cmake
- 😞 Alternative ways to manage toolchain
- 😊 FetchContent

While Meson...

- 😊 Can build only predefined types
- 😊 Can't be tweaked that much
- 😬 Not a complete language
- 😊 crossfile.ini
- 😍 makes it hard to not use standard way
- 😊 subprojects with even cmake support

Meson Integration

Two Ways

NuttX as project's library

Use exported libraries and linker script from NuttX in any other build system.

```
make -C nuttx export
```

Project as app in NuttX

NuttX build system deals with Firmware build.

App integration uses our project's build system and adds results to NuttX linkage.

Phase 1: crossfile

project-apps/meson-apps/Makefile

```
crossfile.ini: .FORCE
    ./crossfile.sh '$(CC)' '$(AR)' '$(CFLAGS)'
```

project-apps/meson-apps/crossfile.sh

```
march="${3/*-march=/}"; march="${march/ */}"
flags_sanit() { sed -E "s#-Wshadow##;s#\\"##g; s#-I #-I#g" <<<"$1" }
cat >crossfile.ini <<-EOF
[binaries]
c = '$1'
ar = '$2'
[built-in options]
c_args = [$(flags_sanit "$cflags" | sed -E "s#([^\ ]+)#\1'#g; s# +#, #g")]
c_link_args = ['-nostdlib', '-static', '-r']
[host_machine]
system = 'nuttx'
cpu_family = '${1%%-*}'
cpu = '${march}'
EOF
```

Phase 1.1: Toolchain.cmake

project-apps/meson-apps/crossfile.sh

```
cat >>crossfile.ini <<-EOF
[properties]
cmake_toolchain_file = '$(pwd)/Toolchain.cmake'
EOF
cat >Toolchain.cmake <<-EOF
add_compile_options($(flags_sanit "$cflags"))
add_link_options(-nostdlib -static -r)
EOF
```

Phase 2: context

project-apps/meson-apps/Makefile

```
context:: build/build.ninja
build/build.ninja: crossfile.ini meson-setup.sh
    ./meson-setup.sh
```

project-apps/meson-apps/meson-setup.sh

```
meson setup --default-library static -Db_staticpic=false \
    --backend ninja --cross-file crossfile.ini build ../..
```

Phase 3: build / build.ninja.mk

project-apps/meson-apps/meson-setup.sh

```
{
  meson introspect --targets "build" |
    jq -r '.[[]
      | select((.type == "static library" or .type == "executable") and .installed)
      | .name, .filename[0], .type' |
    while read -r name && read -r file && read -r type; do
      target="${file#"$sdir/build/"}"
      if [[ "$type" == "executable" ]]; then
        echo "\$(eval \$(call EXECUTABLE,$target,$name,{name//-/_}_main))"
      else
        echo "\$(eval \$(call STATIC_LIBRARY,$target))"
      fi
    done
  meson introspect --buildsystem-files build | jq -r '.[[]' |
    sed 's#^#build/build.ninja: #'
} >"build.ninja.mk"
```

Phase 3.1: built

project-apps/meson-apps/Makefile

```
all:: .built
    @
```

```
.built: build.ninja.mk $(BUILDS)
    touch $@
```

Phase 3.2: build NINJA_TARGET

project-apps/meson-apps/Makefile

```
define NINJA_TARGET

.NOTPARALLEL: build/$(1)
build/$(1): build/build.ninja .FORCE
    flock build ninja $$ (if $$ (V),,--quiet) -C build $(1)

endef
```

Phase 3.3: build STATIC_LIBRARY

project-apps/meson-apps/Makefile

```
define STATIC_LIBRARY
BUILDS += build/$(1).built
LIBRARIES += build/$(1)
$(call NINJA_TARGET,$(1))
build/$(1).built: build/$(1)
    mkdir -p build/$(1).a
    $$ (UNAR) --output build/$(1).a build/$(1)
    cd build/$(1).a && for f in *.o; do mv "$$$$f" "$$(subst
$(DELIM),.,$(1))_$$$f"; done
    $$ (call ARLOCK, $$ (call CONVERT_PATH,$$(BIN)), build/$(1).a/*)
    $$ (call DELDIR, build/$(1).a)
    touch $$@
endef
```

Phase 3.4: libsyms

project-apps/meson-apps/Makefile

```
libsyms: $(LIBRARIES)
    $(NM) --quiet --just-symbols --defined-only $^ | sort -u > $@
```

Phase 3.5: register EXECUTABLE

project-apps/meson-apps/Makefile

```
define EXECUTABLE
```

```
register:: $$$(BUILTIN_REGISTRY)$$$(DELIM)$(2).bdat
```

```
$$$(BUILTIN_REGISTRY)$$$(DELIM)$(2).bdat: build/build.ninja  
    $$$(call REGISTER,$(2),$$$(PRIORITY),$$$(STACKSIZE),$(3))
```

```
endef
```

Phase 3.5: build EXECUTABLE

project-apps/meson-apps/Makefile

```
define EXECUTABLE
    BUILDS += build/$(1).built
    $(call NINJA_TARGET,$(1))
    build/$(1).built: build/$(1) libsyms
        $$ (NM) --quiet --just-symbols --defined-only build/$(1) \
            | sort -u | comm -2 - libsyms >build/$(1).syms
        $$ (OBJCOPY) --redefine-sym main=$(3) \
            --localize-symbols=build/$(1).syms \
            --weaken-symbols=libsyms \
            build/$(1) build/$(1).o
        $$ (call ARLOCK, $$ (call CONVERT_PATH, $$ (BIN)), build/$(1).o)
    touch $$@
endef
```

Demonstration

With LTO we can't modify symbols.

Workaround: add `-flinker-output=nolto-rel` to `c_link_args`

That removes LTO from linked objects objects (executables).

LTO not applied to calls from executables to libraries and nuttx.

LTO is still applied to calls between libraries and from libraries to NuttX

Conclusion

Demonstrated possibilities:

- How to have project specific applications
- How to use off-tree board
- How to convince meson to build for NuttX
- How to merge meson built files to NuttX

All this without forking NuttX or Meson.

Actual implementation provides resiliency against parameters change compared to the examples.

Thank you

Karel Kočí

<https://gitlab.com/Cynerd/nuttx-meson-template/-/tree/master>

<https://gitlab.com/Cynerd/nixos-personal>

<https://git.cynerd.cz/presentations/tree/2026-communitycvercode>