National Aeronautics and Space Administration

CFE CMake Build System

Simplify integrating apps together

CFS “official”

Recycled from other projects

Custom

LC

SC

HK

...

A

B

C

X

Y

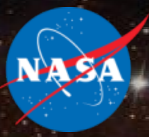
Z

MISSION PROJECT

The CFS concept envisions highly reusable components which are easy to mix and match together, potentially from many different sources. This implies:

- Source packages should be self contained and version-control friendly
- Should be easy to keep up to date with upstream releases of all components
- Should be easy to submit patches to the upstream maintainer if bugs are found

Why make the change?



The classic build has some issues which will make it difficult to use and maintain as CFS moves forward and is (hopefully) adopted by a larger user base.

1. "App Store" distribution model...
   - Missions should be able to pull CFS apps from a wide variety of sources
   - Management ops such as adding an app, removing an app, or upgrading a version of an app within mission project should be as straightforward and automatic as possible
2. Could be more developer friendly...
   - Requires a number hand-edited files / manual operations to manage it
   - Does not reliably handle "deep dependencies" between sources (i.e. dependency rules where a source file #include's another file)
   - Requires putting a lot of "task-specific" data in version-controlled files:
     - Specific target being built for
     - Optimization, debugging, and compiler warning level options
   - Often requires modifying mission-specific source files in place
     - Essentially creates a "fork" which makes it more difficult to upgrade later on, or submit patches back to maintainer if necessary

Classic Makefile Build System

The classic makefiles use a two part design: Outer makefile in “build” directory references inner makefile with specific rules/targets

- Outer part is implemented by project/mission, sets up environment
- Inner part comes with CFS application or upstream source
- Both parts are “tightly coupled”; changes in one do affect the other, but they are *not version controlled together* because they come from different places.
- There were several circumstances of classic build breakage in CFE 6.6 development due to seemingly simple changes such as a file addition
- This also mixes version controlled Makefiles with generated objects

Outer Makefiles
(mission customized)

```
cpu1/cfe
├── cfe-config.mak
├── debug-opts.mak
├── detail_doxy
├── es
│   └── Makefile
├── evs
│   └── Makefile
├── fs
│   └── Makefile
├── link
│   └── Makefile
├── Makefile
├── osal
│   └── Makefile
├── psp
│   └── Makefile
├── sb
│   └── Makefile
├── tbl
│   └── Makefile
├── time
│   └── Makefile
├── unit-test
│   ├── es
```

```
cfe/fsw/cfe-core/src/
├── es
│   └── cfe_es.mak
├── evs
│   └── cfe_evs.mak
├── fs
│   └── cfe_fs.mak
├── make
│   ├── app-rules.mak
│   ├── make-rules.mak
│   └── table-rules.mak
├── sb
│   └── cfe_sb.mak
├── tbl
│   └── cfe_tbl.mak
└── time
    └── cfe_time.mak
```

Inner Makefiles
(provided by applications)

How is CMake different?

CMake is a tool that *generates* the mission-specific Makefiles from a collection of input ingredients

Once prepared, you still use same "make" command to build the project, but avoids a lot of the manual configuration work during the setup and maintenance

- Makes it trivial to switch any of these components "on the fly"
- Intelligent logic can be implemented to ease integration of multiple modules and mission-specific customization
- Follows the "DRY" (don't repeat yourself) mantra
- IDE friendly (Eclipse et al can simply run "make" with no extra environment)

Other advantages

Dependency management:

- The generated makefiles account for deep dependencies across the entire code base and are kept up to date automatically.
- Offers the opportunity for missions to provide “overrides” or customized versions of files
  - Useful for things like tables
  - The customized version can be supplied separately from the original (sample) version
  - Avoids needing to modify the table “in place” which keeps the original source pristine – this is highly advantageous later on when the upstream application evolves and new versions are released.

Development Workflow Efficiency:

- Incremental builds are generally reliable
- Binary files are only built once and re-used whenever possible, which speeds up builds.
  - If two or more CPUs within a mission share the same physical architecture and run the same CFS application or library, the same binary file will be loaded onto both CPUs.
- Quick and Easy to change targets or build/optimizations options on the fly
- Supports multiple builds in parallel (e.g. for different targets) using a single source tree
  - Examples would be a “full simulation” build (for the dev host), a “processor in the loop” test, a “hardware in the loop” test, and a production/release build.

The Ingredients: Mission Specific Config

MISSION SPECIFIC
App selection, Table defintions

BOARD SUPPORT
Cross compiler, Tuning Flags

DEVELOPER PREFERENCES
Optimization, Warning flags

APPLICATION RECIPES
(Part of Apps)

FINAL MAKEFILES

The CFE distribution contains a sample "defs" directory to help. This is intended to be "cloned and owned" and placed at the top level to get the initial setup.

IMPORTANT: This is intentionally ***not*** used in-place in order to keep the CFE source tree "pristine". All modifications/customizations occur to copies outside the original tree.

```
cfe/cmake/sample_defs
├── cpu1_cfe_es_startup.scr
├── cpu1_msgids.h
├── cpu1_platform_cfg.h
├── default_osconfig.h
├── sample_mission_cfg.h
├── sample_perfids.h
├── targets.cmake
├── toolchain-arm-cortexa8_neon-linux-gnueabi.cmake
├── toolchain-cpu1.cmake
├── toolchain-cpu2.cmake
├── toolchain-cpu3.cmake
├── toolchain-i686-rtems4.11.cmake
└── toolchain-powerpc-440_softfp-linux-gnu.cmake
```

Directory Structure: Configuration Files

Sample Configuration Directory

```
sample_defs
├── cpu1_cfe_es_startup.scr
├── cpu2_cfe_es_startup.scr
├── cpu3_cfe_es_startup.scr
├── default_osconfig.h
├── default_platform_cfg.h
├── eds
├── sample_mission_cfg.h
├── sample_perfids.h
├── targets.cmake
├── toolchain-i686-rtems4.11.cmake
└── toolchain-powerpc-7400-poky-linux.cmake
```

- cpu1_cfe_es_startup.scr: CPU-specific startup scripts (copied as "cfe_es_startup.scr" on target)
- default_osconfig.h: OSAL configuration #defines (wrapped as "osconfig.h" per OSAL build)
- default_platform_cfg.h: CFE Platform configuration #defines (customizable per CPU, may be combined, wrapped as cfe_platform_cfg.h per CFE build)
- sample_mission_cfg.h: Top-level mission configuration (wrapped as cfe_mission_cfg.h for build)

targets.cmake: global CPU target configuration file
Sample snippet (repeat for all CPUs)

```
SET(TGT1_NAME cpu1)
SET(TGT1_SYSTEM powerpc-7400-poky-linux)
SET(TGT1_PLATFORM default)
SET(TGT1_APPLIST sample_app ci_lab to_lab)
SET(TGT1_FILELIST cfe_es_startup.scr)
```

- TGT1_NAME: Name of CPU (free form)
- TGT1_SYSTEM: By default, cross compile using instructions from powerpc-7400-poky-linux toolchain file
- TGT1_PLATFORM: "cfe_platform_cfg.h" will be a wrapper of "default_platform_cfg"
- TGT1_APPLIST: List of CFS Applications to build and install
- TGT1_FILELIST: Single file to be copied into installation tree (straight copy; prefixed with "<cpuname>_")

The Ingredients: Board Support

MISSION SPECIFIC
App selection, Table defintions

BOARD SUPPORT
Cross compiler, Tuning Flags

DEVELOPER PREFERENCES
Optimization, Warning flags

APPLICATION RECIPES
(Part of Apps)

FINAL MAKEFILES

The cross compiler to use is specified via a "toolchain file". These are the files in the defs directory beginning with a `toolchain-` prefix.

- The toolchain file specifies an initial set of information, such as which cross compiler to use, and the minimal set of flags (usually `-m`, `-f` options) to build for the CPU.
- It also indicates which PSP and OSAL to use for the target
- The PSP and OSAL, in turn, can add additional CFLAGS for tuning (usually `-D` options like "`XOPEN_SOURCE=600`" for GNU/Linux platforms)

The Ingredients: Developer Preferences

MISSION SPECIFIC
App selection, Table defintions

BOARD SUPPORT
Cross compiler, Tuning Flags

DEVELOPER PREFERENCES
Optimization, Warning flags

APPLICATION RECIPES
(Part of Apps)

FINAL MAKEFILES

Additional ingredients are sourced from the environment at "setup time"

- These are *not* stored in any version controlled file, as it may change from build to build. But they *are* cached in the generated files, so it is only specified at first setup.
- Can override all targets by setting "`SIMULATION`" variable. For instance, setting "`SIMULATION=native`" builds all code for the dev host, overriding the default cross compiler selection. Useful for debug builds!
- "`CMAKE_BUILD_TYPE`" can be "debug" (usually means `-O0`) or "release" (usually means `-O3`) but actual effect depends on specific BSP (compiler/tool) selections.
- "`OSAL_USER_C_FLAGS`" is useful for setting warning options (e.g. `-Wall -Werror`)

The Ingredients: Applications

MISSION SPECIFIC
App selection, Table defintions

BOARD SUPPORT
Cross compiler, Tuning Flags

DEVELOPER PREFERENCES
Optimization, Warning flags

APPLICATION RECIPES
(Part of Apps)

FINAL MAKEFILES

The final ingredients come as part of each application / library / component that has been merged into the mission source tree.

- Each CFS app should come with its own recipe that says what the app binary file(s) are and what source(s) go into each one
- Each recipe should also indicate what its dependencies are, if any, so the proper set of "`-I`" options can be put onto the command line.
  - The absolute location of all modules is indicated in a CMake variables by the convention `${<NAME>_MISSION_DIR}` where <NAME> refers to the module.
  - For instance "`include_directories(${cfs_lib_MISSION_DIR}/inc)`" would add the include path for CFS LIB.

National Aeronautics and Space Administration

Directory Structures

Example Project Layout

Top Level Directory

```
work
├── apps
├── build
├── cfe
├── Makefile
├── osal
├── psp
├── README.md
├── sample_defs
└── tools
```

CFS Application Source Code
One subdirectory per application, which directly maps to upstream/official source (not modified locally in any way)

Build Output and intermediate files
(100% generated files; not version controlled)

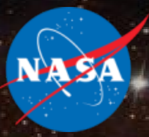
Subtrees containing "upstream" sources
Should stay unmodified from their original state; no editing or customizing to the target system in these files

Mission configuration
Contains "overrides" or extensions to the distribution files that provide mission-specific needs

All "mission-specific" information is consolidated in the `_defs` directory, such as:

- Number and type/architecture of processor boards in use, and the corresponding `cfe_platform_cfg.h` file for each processor board
- CFS Applications to compile for each processor board
- Top-level mission configuration (`cfe_mission_cfg.h`)
- Table definitions
- Script files

BSP/OS option sources



Top Level Directory

```
psp/fsw/pc-linux/make
├── build_options.cmake
├── compiler-opts.mak
└── link-rules.mak
osal/src/os/posix
├── build_options.cmake
├── osal.mak
├── osapi.c
├── osfileapi.c
├── osfilesys.c
├── osloader.c
├── osnetwork.c
└── ostimer.c
```

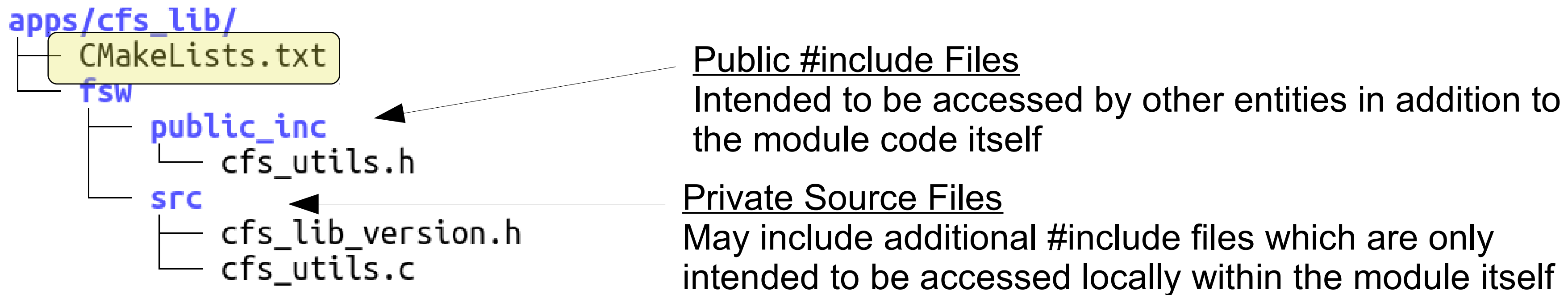
PSP Options
Specifies flags such as `-D_LINUX_OS_`

OSAL Options
Specifies flags such as `-D_XOPEN_SOURCE=600` and additional system libraries for linking

Each PSP and/or OS support layer may add additional flags to the build.

It is important to note these are really “scripts” that are executed at setup time, so they can contain more advanced logic if need (conditionals, loops, subroutines etc) but are usually just a few “set” statements.

Example Application/Library Directory



Example CmakeLists.txt Content

```
# Reference include files in the fsw/public_inc directory
include_directories(fsw/public_inc)

# Get a list of all source files in the fsw/src directory
aux_source_directory(fsw/src APP_SRC_FILES)

# Create the app module
add_cfe_app(cfs_lib ${APP_SRC_FILES})

# Specify that this object depends on the system math library (m)
target_link_libraries(cfs_lib m)
```

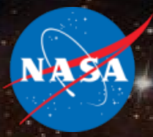
Directory Structure: Build Tree

This is generated during the “prep” phase. Consider a theoretical mission configuration with 3 CPUs:

- “cpu1” runs PowerPC/Linux and is the main processor, running CI/TO
- “cpu2” and “cpu3” run RTEMS as helpers for data collection, runs only sample_app

```
build
├── doc
├── i686-rtems4.11          <-- Subdirectory for all objects built using "i686-rtems4.11" toolchain
│   │                           (2 CPUs using RTEMS here)
│   ├── apps
│   │   └── sample_app      <-- App object files built using cross compiler – just 1 here
│   ├── cfe_core_default    <-- CFE core build – shared by both CPUs since they share the same config
│   │   └── inc
│   ├── cpu2                <-- Final "exe" file for each actual target
│   ├── cpu3
│   ├── inc
│   ├── obj
│   ├── osal                <-- OSAL build for RTEMS
│   └── psp
│       └── pc-rtems        <-- PSP build for pc-rtems
├── inc
├── obj
├── powerpc-7400-poky-linux <-- Subdirectory for all files objects using "powerpc-7400-poky-linux" toolchain
│   ├── apps                <-- App object files built using cross compiler – all 3 here
│   │   ├── ci_lab              (note sample_app is built once for each arch)
│   │   ├── sample_app
│   │   └── to_lab
│   ├── cfe_core_default    <-- CFE core build for this target
│   │   └── inc
│   ├── cpu1                <-- Final "exe" file for this target
│   ├── inc
│   ├── obj
│   ├── osal                <-- OSAL build for POSIX
│   └── psp
│       └── pc-linux        <-- PSP build for pc-linux
└── src
```

Directory Structure: Scoped include files

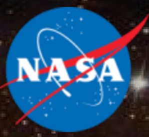


Note that the wrapped `#include` files are put in scope-specific locations.
Include files apply to the entire mission, a specific architecture, or a specific platform.

```
build
├── doc
├── i686-rtems4.11
│   ├── apps
│   │   └── sample_app
│   ├── cfe_core_default
│   │   └── inc
│   ├── cpu2
│   ├── cpu3
│   ├── inc
│   ├── obj
│   ├── osal
│   └── psp
│       └── pc-rtems
├── inc
├── obj
├── powerpc-7400-poky-linux
│   ├── apps
│   │   ├── ci_lab
│   │   ├── sample_app
│   │   └── to_lab
│   ├── cfe_core_default
│   │   └── inc
│   ├── cpu1
│   ├── inc
│   ├── obj
│   ├── osal
│   └── psp
│       └── pc-linux
└── src
```

- i686-rtems4.11 / cfe_core_default / inc: #include files with **platform** scope (e.g. “cfe_platform_cfg.h”) (Generally not available outside cfe-core)
- i686-rtems4.11 / inc: #include files specific to this architecture (e.g. osconfig.h for RTEMS) (Not available outside this architecture)
- build / inc: #include files with **mission** scope (i.e. cfe_mission_cfg.h)
- powerpc-7400-poky-linux / cfe_core_default / inc: #include files with **platform** scope (e.g. “cfe_platform_cfg.h”) (Generally not available outside cfe-core)
- powerpc-7400-poky-linux / inc: #include files specific to this architecture (e.g. osconfig.h for Linux) (Not available outside this architecture)

Scoped include files - caveats



Historically, applications would do:

```
#include "cfe_platform_cfg.h"
```

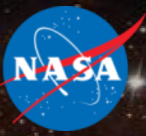
This practice is now deprecated going forward – portable applications *should not* be dependent on a specific CFE platform configuration.

- May introduce subtle/undetectable ABI incompatibilities if the binary is loaded on a platform using different platform config values (size of objects could change).
- In many circumstances this `#include` is not really even used
- In other cases it is only for extra "verification check" purposes
  - But any such verification check is making assumptions about how CFE works internally, which may change from version to version or system to system.
  - Case in point: "`CFE_SB_HIGHEST_VALID_MSGID`" – assumptions made in apps will probably be wrong for CCSDS version 2 configurations
  - The rule: only CFE SB (which owns the implementation) should verify message IDs, via the CFE SB API (e.g. `CFE_SB_ValidateMsgId()` call)

BUT – since many apps were already doing this, there is a workaround in place:

- If an toolchain config is only used on a **single CPU**, then the inclusion is allowed
- This should be removed by CFE 6.7 once apps are cleaned up

Summary



The CMake build system is intended to ease app integration and further enhance the CFS goals of source module portability.

Although this does impact the source to some degree (e.g. stricter scoping of #include files), implementing those changes should ultimately improve the modularity of the software in the long run.