cFS Applications in Rust with n2o4

Zachary Catlin

Dept. of Astronomy and Astrophysics, Penn State Univ.

PennState

Notice

This presentation does not include any material restricted by US export control laws.

PennState

The team:

- Abe Falcone, Principal Investigator
- Michael Betts
- Jacob Buffington
- Zachary Catlin
- Joseph Colosimo
- Timothy Emeigh
- Derek Fox
- Hannah Grzybowski
- Fredric Hancock
- Evan Jennerjahn
- Jordan Josties
- David Palmer (LANL)
- Lukas Stone
- Ian Thornton
- Mitchell Wages
- Daniel Washington
- Michael Zugger
- *and several alumni*

Los Alamos NATIONAL LABORATORY

PennState

The team:

- Abe Falcone, Principal Investigator
- Michael Betts
- Jacob Buffington
- Zachary Catlin
- Joseph Colosimo
- Timothy Emeigh
- Derek Fox
- Hannah Grzybowski
- Fredric Hancock
- Evan Jennerjahn
- Jordan Josties
- David Palmer (LANL)
- Lukas Storre
- Ian Thornton
- Mitchell Wages
- Daniel Washington
- Michael Zugger
- *and several alumni*

Presenting Thursday morning!

Los Alamos NATIONAL LABORATORY

PennState

The team:

(Note: latest available group photo, with a slightly different set of people)

PennState

Context: the BlackCAT mission

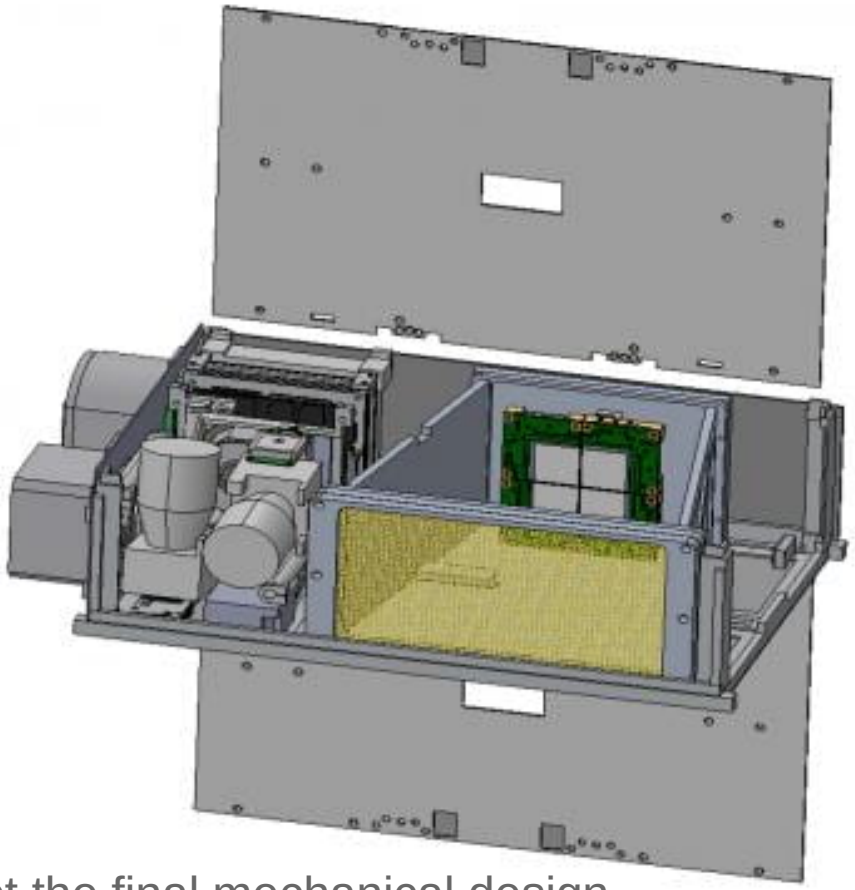
Astronomy... *IN SPACE!*

BlackCAT

PennState

BlackCAT

- Soft X-ray coded aperture telescope using novel hybrid CMOS detectors
- Detects and localizes astronomical transients in the ~0.3–20 keV band for rapid follow-up by other facilities
- ~1 sr field of view, pointed anti-sun
- Sole payload on a 6U CubeSat in a ~550-km sun-synchronous orbit
- Expected launch date: late 2024

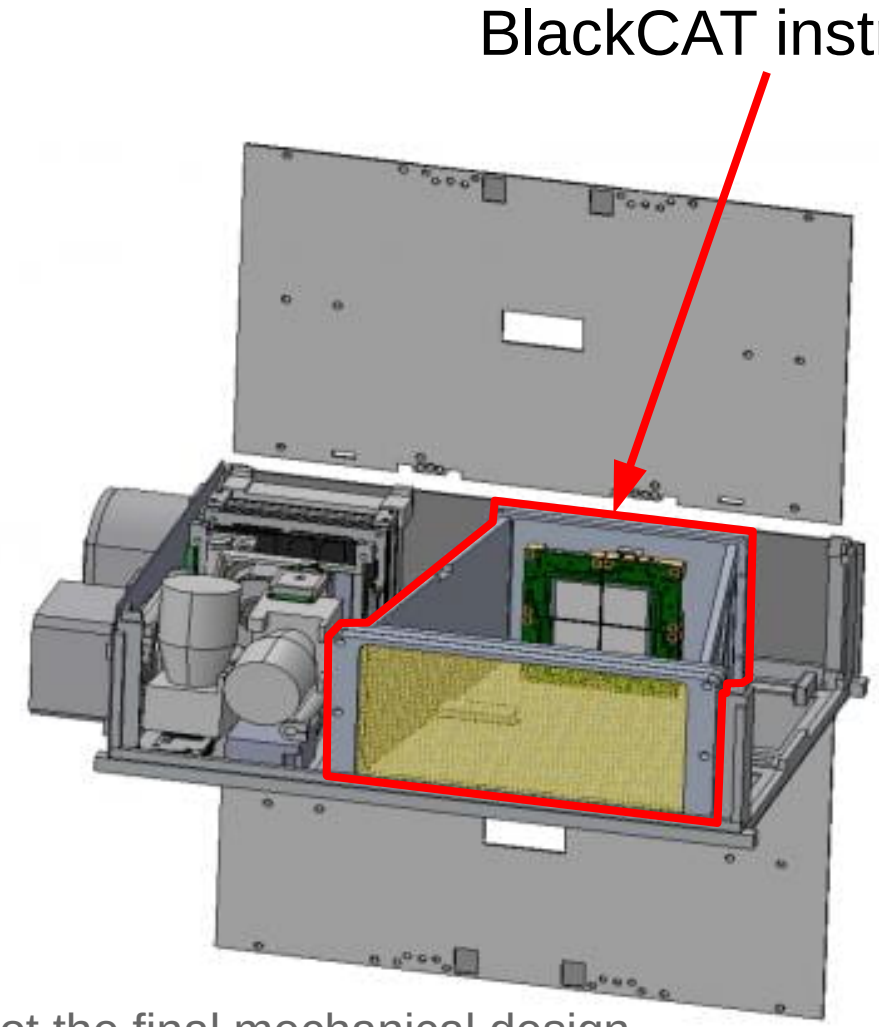


Note: not the final mechanical design, but should be close

PennState

BlackCAT

- Soft X-ray coded aperture telescope using novel hybrid CMOS detectors
- Detects and localizes astronomical transients in the ~0.3–20 keV band for rapid follow-up by other facilities
- ~1 sr field of view, pointed anti-sun
- Sole payload on a 6U CubeSat in a ~550-km sun-synchronous orbit
- Expected launch date: late 2024

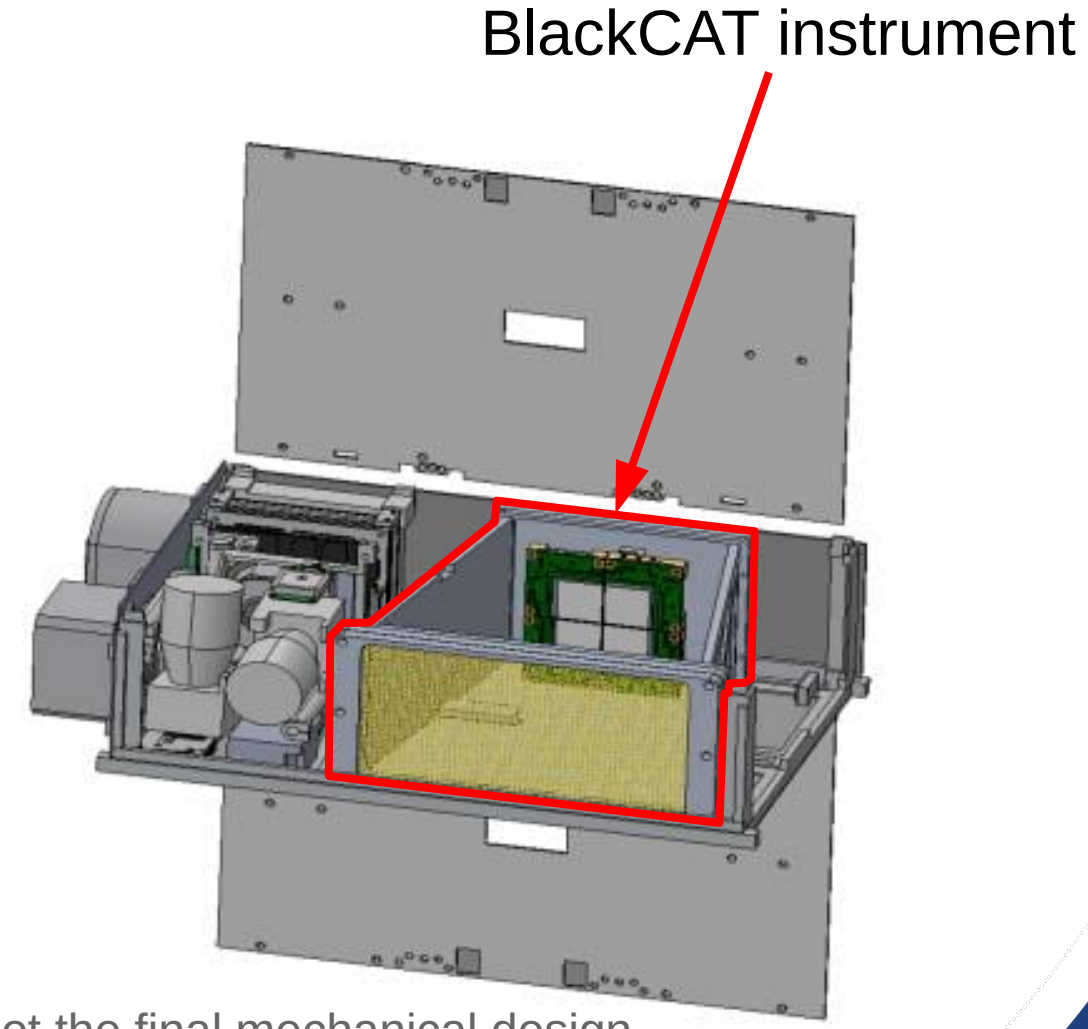


Note: not the final mechanical design, but should be close

PennState

BlackCAT

- Instrument hardware/ gateware/software and science ops provided by the BlackCAT team (PSU/LANL)
- Spacecraft bus, non-instrument avionics, and ground station provided by NanoAvionics
- Sensor development by PSU and Teledyne Imaging Systems
- Mission and sensor dev. funding by NASA



Note: not the final mechanical design, but should be close

PennState

Important BlackCAT flight software requirements

- Needs to be able to enable, disable, and configure each of the four detectors, and analyze their output
- Needs to be able to recognize probable interesting transients (gamma-ray bursts, etc.) within seconds and localize their position on the sky
- Needs to be able to send notifications of transients to ground-side systems in near real time (~1–3 min delay)
- Needs to send (during scheduled ground-station passes) X-ray photon events around the time of transients (stretch goal: and all other times as well)

PennState

BlackCAT flight software environment

- Instrument computer: Xiphos Q7S
- Zynq-7020: 2 Cortex-A9 cores at ~700 MHz + FPGA fabric
- 256 MiB ECC DRAM
- Operating system: Linux (Yocto-based distribution w/ Xiphos customizations)
- Flight software framework: Core Flight System (cFS)
- BlackCAT peripherals:
  - 4 TIS Speedster-EXD 550 detectors
  - DACs and PWM for power supplies
  - Instrument health: voltage monitors, temperature sensors, heaters
  - RS-422 serial to spacecraft avionics

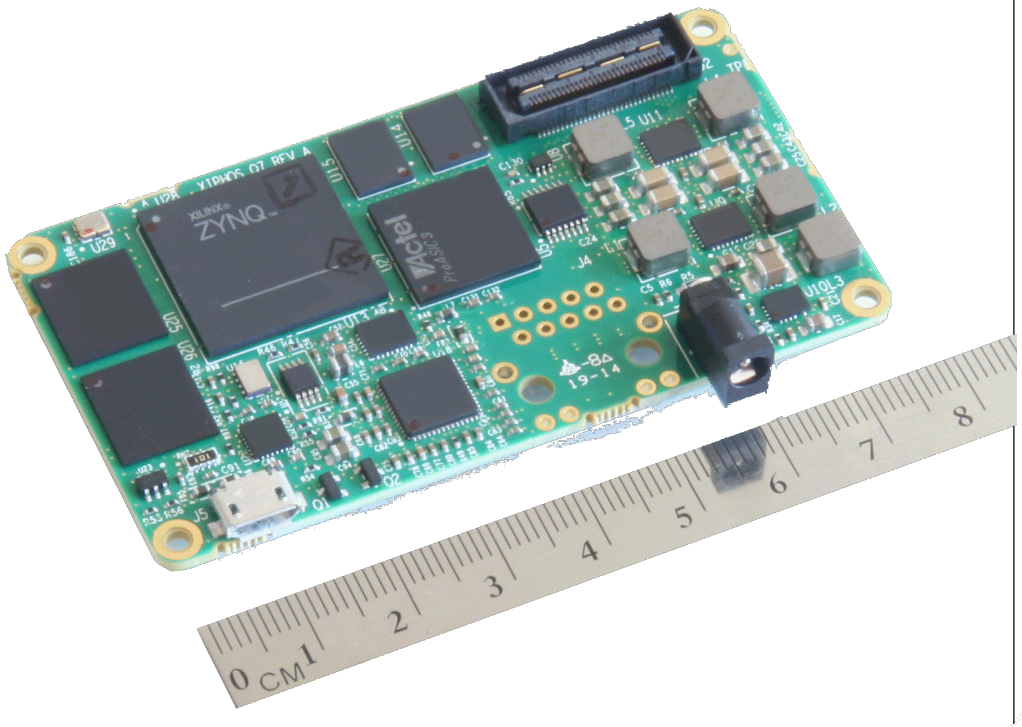


Image credit: Xiphos Systems Corp.

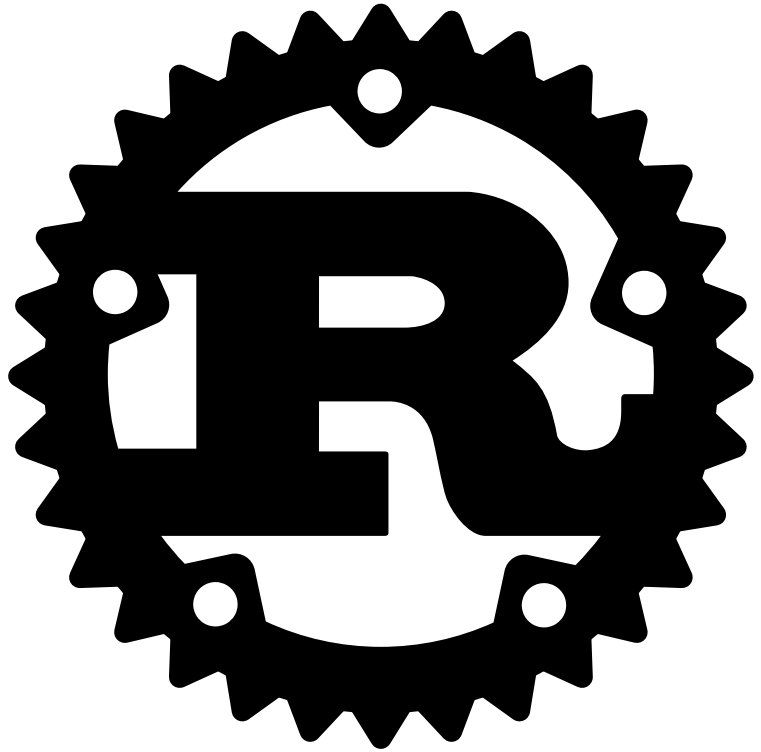
PennState

The Rust programming language

Or: how I learned to stop worrying and love the borrow checker

Rust

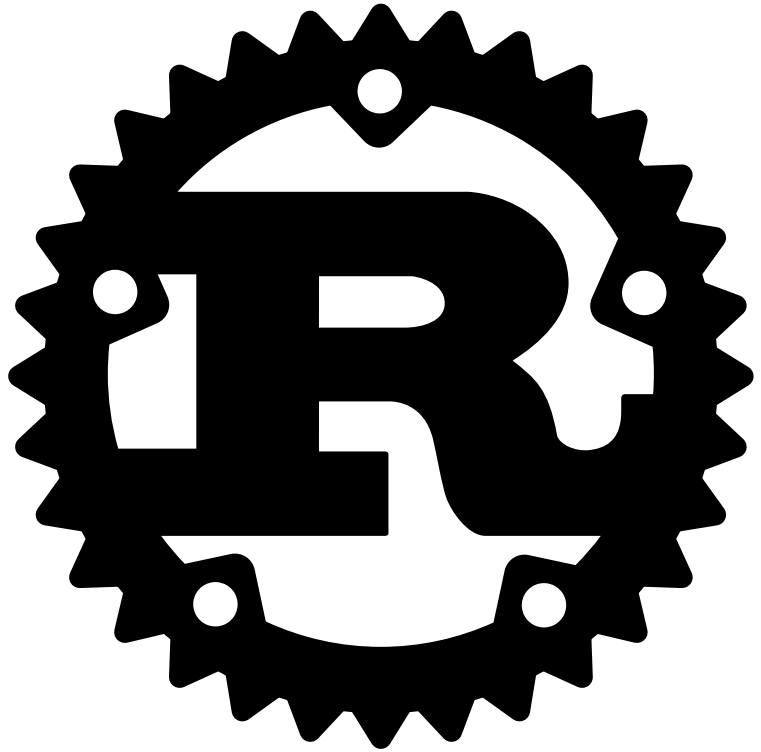
- Rust is a systems programming language
- “helps you write faster, more reliable software”
  —Introduction, *The Rust Programming Language*
- Uses an ownership system and reference lifetimes to ensure memory and thread safety by default
- Incorporates concepts from higher-level languages when they impose little or no runtime overhead
- *De facto* standard compiler, `rustc`, outputs fast native code (using language-specific optimization, followed by LLVM)



PennState

Rust history, very briefly

- 2006: started by Graydon Hoare as a personal project
- 2010: made public after Mozilla took interest
- 2015: language stabilized enough for 1.0 release
- present: under active development, but with stability guarantees post-1.0
  - separate `stable` and `nightly` channels
- Used within Firefox
- Used in production by Google, AWS, etc., etc.



PennState

Language characteristics

- C-esque syntax
- Few new language concepts (but sometimes the first popular language with the concept)
- Expression-based
- Variables immutable by default
- Strongly, statically-typed, but with type inference
- Memory-safe by default (but with `unsafe` keyword for temporary exceptions)

```
fn an_operation(a: u32, b: u8) -> u32 {
    let x = match (b, a) {
        (0, a) => a % 2,
        (1, _) => 42,
        (_, a) => {
            let a = (a % 2);
            a + (b as u32)
        }
    };

    x + 3
}
```

Language characteristics

- Product (`struct`) and sum (`enum`) types
- Generics for types and functions/methods
- No object-oriented inheritance, but traits available for behaviors generic over certain types
- References: pointers, but with additional semantics around mutability, lifetimes; never `NULL`!
- Much more!

```
struct A {
    fld1: u32,
    fld2: bool,
    fld3: Option<i32>,
}

enum B {
    CaseA,
    CaseB(A),
    CaseC,
}

trait MyOperation {
    fn op(&self) -> bool;
}

struct GenericStruct<'a, T: MyOperation> {
    subject: T,
    field_x: &'a B,
    field_y: A,
}
```

PennState

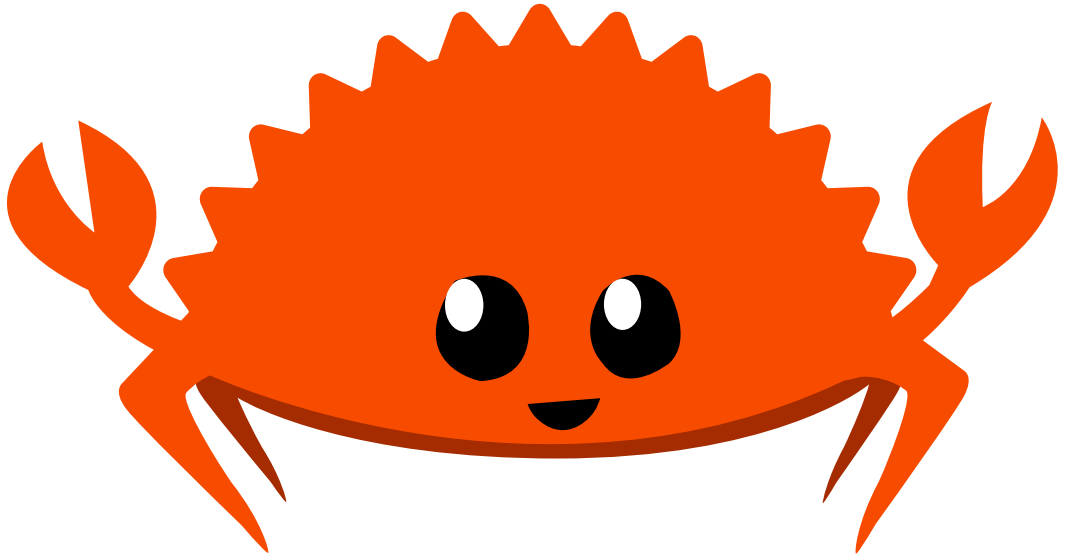
Default tooling

- Rustup: toolchain downloader/updater
- Cargo: standard package manager & build system
  - Crates.io: standard repository of open-source Rust crates
  - Build scripts: build-time code generation and customization for environment
- Rustdoc: generator of API documentation

PennState

Evaluation

- I like it!
- Not perfect, but an improvement on C
- *Does* have a learning curve
- Generally, where language is complex, difficult, or just *different*, it is for good reasons
- Language has good ergonomics, a good compiler, good tooling, and good documentation



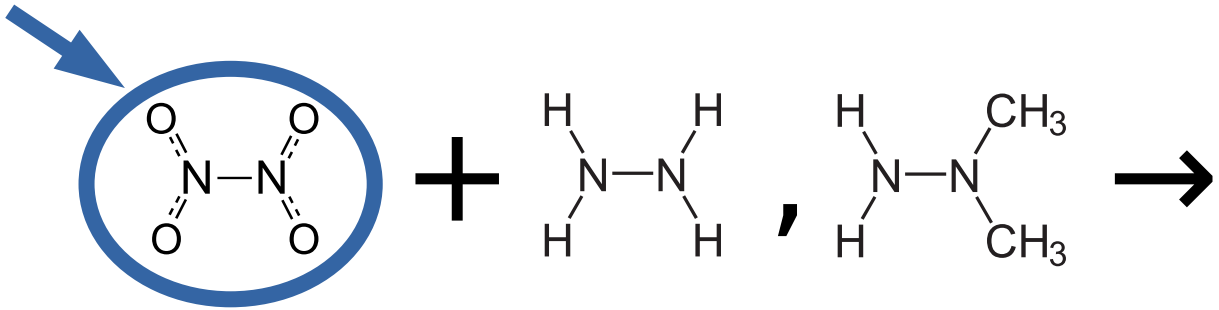
Ferris, the unofficial mascot of Rust

PennState

cFS apps in Rust, how do I even

Rust bindings to cFS API functions

- To be a cFS application, we should use cFS API functions
- Rust can call out to C functions (in `unsafe` code)… but raw function calls aren't idiomatic in this case.
- Solution: create lightweight wrappers that provide a Rustic façade
- We call it `n2o4`.



PennState

Image credits: Wikipedia contributors; NASA

API definitions: rewrite it in Rust… *automatically*

- *Problem*: Rust doesn’t natively read C header files
- *Solution*: use the `bindgen` crate in a build script!

build.rs (simplified)

```
extern crate bindgen;

fn main() {
    let bindings = bindgen::builder()
        .header("cfs-all.h")
        .allowlist_type("(CFE|OS|OSAL|CCSDS).*")
        [...]
        .generate()
        .expect("Unable to generate bindings");
    bindings
        .write_to_file("${OUT_DIR}/cfs-all.rs");
}
```

cfs-all.h (excerpt)

```
#include <cfe.h>
#include <osapi.h>

#include <cfe_es_msg.h>
#include <cfe_evs_msg.h>

[...]
```

PennState

API definitions: rewrite it in Rust… *almost automatically*

- *Problem*: Rust doesn’t natively read C header files
- *Solution*: use the `bindgen` crate in a build script!
- …and compile a small C file with wrappers for `static inline` functions

build.rs (simplified)

```
extern crate bindgen;
extern crate cc;

fn main() {
    let bindings = bindgen::builder()
        .header("cfs-all.h")
        .header("cfs-shims.h")
        .allowlist_type("(CFE|OS|OSAL|CCSDS).*")
        [...]
        .generate()
        .expect("Unable to generate bindings");
    bindings
        .write_to_file("${OUT_DIR}/cfs-all.rs");

    cc::Build::new()
        .file("cfs-shims.c")
        .compile("cfs-shims");
}
```

cfs-shims.c (excerpt)

```
#include <cfe.h>

[...]

CFE_SB_MsgId_Atom_t SHIM_CFE_SB_MsgIdToValue(
    CFE_SB_MsgId_t MsgId
) {
    return CFE_SB_MsgIdToValue(MsgId);
}

[...]
```

PennState

Wrapping it up: simple example

- cfs-all.rs now has a bunch of usable definitions, but not as a safe, idiomatic Rust interface
- So we write small, safe wrappers
- Often the wrappers will be inlined completely for zero runtime overhead

${OUT_DIR}/cfs-all.rs

```
[...]
pub type CFE_ES_RunStatus = ::core::ffi::c_uint;
pub const
CFE_ES_RunStatus_CFE_ES_RunStatus_APP_RUN:
CFE_ES_RunStatus = 1;
pub const
CFE_ES_RunStatus_CFE_ES_RunStatus_APP_EXIT:
CFE_ES_RunStatus = 2;
pub const
CFE_ES_RunStatus_CFE_ES_RunStatus_APP_ERROR:
CFE_ES_RunStatus = 3;
[...]

extern "C" {
    pub fn CFE_ES_ExitApp(ExitStatus: uint32);
}

[...]
```

PennState

Wrapping it up: simple example

- cfs-all.rs now has a bunch of usable definitions, but not as a safe, idiomatic Rust interface
- So we write small, safe wrappers
- Often the wrappers will be inlined completely for zero runtime overhead

src/cfe/es.rs

```
/// The status (or requested status)
/// of a cFE application.
#[repr(u32)]
pub enum RunStatus {
    AppError
        = CFE_ES_RunStatus_CFE_ES_RunStatus_APP_ERROR,
    AppExit
        = CFE_ES_RunStatus_CFE_ES_RunStatus_APP_EXIT,
    AppRun = CFE_ES_RunStatus_CFE_ES_RunStatus_APP_RUN,
    [...]
}
[...]

/// Exits from the current application.
#[inline]
pub fn exit_app(exit_status: RunStatus) -> ! {
    unsafe { CFE_ES_ExitApp(exit_status as u32) };

    // If we get here, something's gone wrong with cFE:
    unreachable!("CFE_ES_ExitApp returned, somehow");
}
```

Wrapping it up: simple example

- cfs-all.rs now has a bunch of usable definitions, but not as a safe, idiomatic Rust interface
- So we write small, safe wrappers
- Often the wrappers will be inlined completely for zero runtime overhead

user of n2o4

```
use n2o4::cfe::es;

[...]

if unrecoverable_error() {
    es::exit_app(es::RunStatus::AppError);
}
```

Observations on cFS APIs from a Rust perspective

- Handles make for nice, easy-to-wrap abstractions
- Obeying temporal restrictions on pointer accesses can be enforced statically

User of n2o4

```
use n2o4::cfe::sb::{Pipe, TimeOut};

let mut p: n2o4::cfe::sb::Pipe = [...];

p.receive_buffer(TimeOut::Forever, |msg_maybe| {
    if let Ok(msg) = msg_maybe {
        [...process message...]
    }
});
```

src/cfe/sb.rs

```
/// A software bus pipe.
pub struct Pipe {
    /// cFE ID for the pipe.
    pub(crate) id: CFE_SB_PipeId_t,
}

impl Pipe {
    #[inline]
    pub fn receive_buffer<T, F>(
        &mut self,
        time_out: TimeOut,
        closure: F
    ) -> T
    where
        F: for<'a> FnOnce(Result<&'a Message, Status>) -> T,
    {
        [...]
        let s: Status = unsafe {
            CFE_SB_ReceiveBuffer(&mut buf_ptr,
                self.id, time_out.into())
        }.into();
        [...]
    }

}
```

PennState

Observations on cFS APIs from a Rust perspective

- Even things like printf(3) format strings and their use can be type-checked at compile time without special compiler support

```
use core::ffi::c_char;
use n2o4::cfe::evs::{
    EventSender, EventType::Information
};
use printf_wrap::PrintFmt;

const FMT: PrintfFmt<(u32, c_char)>
    = PrintfFmt::new_or_panic("A: %x, B: %c\0"); // OK
const BAD_FMT: PrintfFmt<(u32)>
    = PrintfFmt::new_or_panic("%s %s %s\n\0");    // compile
                                                  // error

[...]

fn do_a_thing(ev: &EventSender) {
    [...]
    ev.send_event2(4, Information, FMT,  // OK
        5u32, b'x' as c_char
    );
    [...]
    ev.send_event2(4, Information, FMT,  // compile
        5u32, 42u32                      // error
    );
    [...]
}
```

PennState

But wait...

What about actually integrating into the cFS build system?

PennState

Building a Rust-based cFS app

CMakeLists.txt

```
project(CFE_FOO_APP C)

add_cfe_app(foo)
```

rust-fsw/Cargo.toml

```
[package]
name = "foo"
version = "0.0.0"
edition = "2021"

[dependencies]
n2o4 = {
  git = "https://github.com/BlackCAT-CubeSat/n2o4.git",
  rev = "1ad09b2dbbca8687bc8a710cfccd4e7e5d78952e"
}
```

rust-fsw/src/lib.rs

```
#![no_std]

use n2o4::cfs::{es, evs, sb};

/// Entry point of application.
pub fn foo_APP_MAIN() {
    [...]
}
```

Doesn't "just" work. Need to integrate build systems.

PennState

Building a Rust-based cFS app

CMakeLists.txt

```
project(CFE_FOO_APP C)

add_cfe_app(foo fsw/src/placebo.c)

set(RUST_TARGET "armv7-unknown-linux-gnueabihf")

set(RUST_SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/rust-fsw)
set(CARGO_TARGET_DIR ${CMAKE_CURRENT_BINARY_DIR}/target)
set(LIB_BUILD_DIR ${CARGO_TARGET_DIR}/${RUST_TARGET}/release)
set(LIB_FILE ${LIB_BUILD_DIR}/libfoo.a)

add_custom_command(
  OUTPUT ${LIB_FILE}
  WORKING_DIRECTORY ${RUST_SOURCE_DIR}
  COMMAND ${CMAKE_COMMAND} -E env
    "RUST_CFS_SYS_COMPILE_DEFINITIONS=[...]"
    "RUST_CFS_SYS_INCLUDE_DIRECTORIES=[...]"
    "RUST_CFS_SYS_COMPILE_OPTIONS=[...]"
    "CFLAGS=[...]"
    "CRATE_CC_NO_DEFAULTS=true"
    "BINDGEN_EXTRA_CLANG_ARGS=[...]"
    cargo +nightly build --jobs 1 -Z build-std=std,panic_abort
    --release --target ${RUST_TARGET} --target-dir ${CARGO_TARGET_DIR} --quiet
  DEPFILE ${LIB_BUILD_DIR}/libfoo.d
  DEPENDS ${RUST_SOURCE_DIR}/Cargo.toml
  VERBATIM
)

add_custom_target(foo_rust_build DEPENDS ${LIB_FILE})

add_library(foo_rust_lib STATIC IMPORTED)
add_dependencies(foo_rust_lib foo_rust_build)
set_target_properties(foo_rust_lib
  PROPERTIES
  IMPORTED_LOCATION ${LIB_FILE}
)

target_link_libraries(foo foo_rust_lib m)

target_link_options(foo
  PUBLIC LINKER:--require-defined=foo_APP_MAIN
)

set_directory_properties(
  PROPERTIES
  ADDITIONAL_CLEAN_FILES ${CARGO_TARGET_DIR}
)
```

rust-fsw/Cargo.toml

```
[package]
name = "foo"
version = "0.0.0"
edition = "2021"

[lib]
crate-type = ["staticlib"]

[dependencies]
n2o4 = { [...] }

[profile.release]
panic = "abort"
```

fsw/src/placebo.c

```
const char placebo = 'a';
```

rust-fsw/src/lib.rs

```
#![no_std]

use n2o4::cfs::{es, evs, sb};

/// Entry point of application.
#[no_mangle]
pub extern "C"
fn foo_APP_MAIN() {
    [...]
}

#[panic_handler]
fn panic([...]) -> ! {
  es::exit_app(
    es::RunStatus::AppError
  );
}
```

Integration can be done... with a lot of stitching...

PennState

Building a Rust-based cFS app

CMakeLists.txt

```
project(CFE_FOO_APP C)

# Assuming rust_cfs_app.cmake is included from
# arch_build_custom.cmake, and RUST_TARGET
# and a couple other variables are set in
# toolchain-*.cmake:

add_cfe_app(foo fsw/src/placebo.c)

cfe_rust_crate(foo foo)

target_link_options(foo
  PUBLIC LINKER:--require-defined=foo_APP_MAIN
)
```

rust-fsw/Cargo.toml

```
[package]
name = "foo"
version = "0.0.0"
edition = "2021"

[lib]
crate-type = ["staticlib"]

[dependencies]
n2o4 = { [...] }

[profile.release]
panic = "abort"
```

fsw/src/placebo.c

```
const char placebo = 'a';
```

rust-fsw/src/lib.rs

```
#![no_std]

use n2o4::cfs::{es, evs, sb};

/// Entry point of application.
#[no_mangle]
pub extern "C"
fn foo_APP_MAIN() {
    [...]
}

#[panic_handler]
fn panic([...]) -> ! {
  es::exit_app(
    es::RunStatus::AppError
  );
}
```

*...**much** of which can be wrapped for easy re-use.*

PennState

Conclusions, and an invitation

Conclusions

- Rust is pretty good
- You *can* write cFS applications in Rust
  ...with a fair bit of non-default setup
  - and currently only with the `nightly` channel
- So far, application development has justified building this infrastructure

Invitation to join in

- n2o4 and the build support are a work in progress, and you can help make it better!
  - Bindings for more cFE, OSAL APIs
  - Better testing support
  - API version flexibility
  - Building for non-Linux targets
  - Cargo build concurrency
  - …
- We're open to questions, pull requests, issues, etc.
- Or just use what we've made so far!

https://github.com/BlackCAT-CubeSat/n2o4

PennState

Questions

Zachary Catlin
zec0@psu.edu
zec