# An End-to-End FSW Development Approach

## Using MicroPython and the Basilisk Software Testbed

Mar Cols Margenet[*], Hanspeter Schaub[†] and Scott Piggott[‡]

[*]Graduate Researcher, University of Colorado
[†]Professor, Glenn L. Murphy Chair, University of Colorado
[‡]ADCS Integrated Simulation Software Lead, Laboratory for Atmospheric and Space Physics

Ann and H. J. Smead Aerospace
Engineering Sciences Department
University of Colorado, **Boulder**

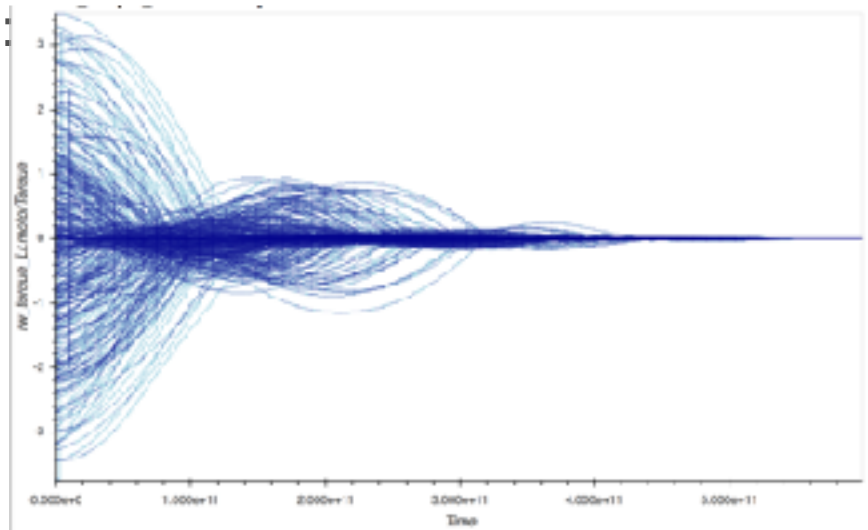
# Motivation

- **FSW testing in different environments…**
  - **Desktop** testbed environment
  - **Embedded** testbed environment: hardware flight processor or emulated
- **Gap between environments** implies there's a **migration effort.**
- **Desired FSW Development approach**:
  - Keep both testbeds while minimizing migration effort
  - Algorithm source code remains unchanged: **"Test what fly, fly what you test"**.
  - **Desktop dev proposal**: Python user-interface and C/C++ algorithm source code
  - **Embedded dev proposal 1**: CFS middleware and C/C++ algorithm source code
  - **Embedded dev proposal 2:** MicroPython user-interface and C/C++ source code

# Desktop Development Proposal

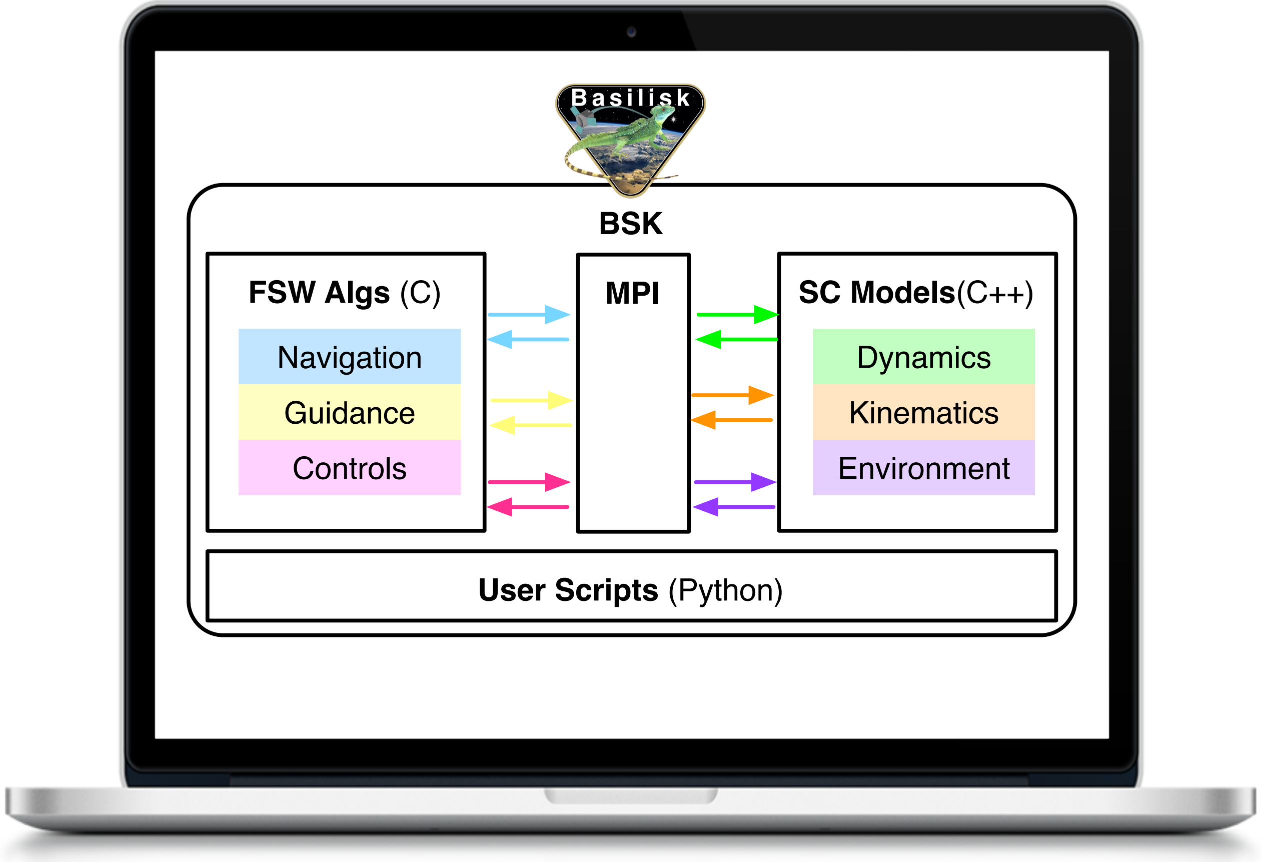
- **Proposal:** Python user-interface with underlying C/C++  flight source code
- **Python pro's:** high-level language with **powerful features** and **large community**
- **Python con's:** runtime insufficiently well-controlled for FSW time-critical applications
- **Python**: let's take a closer look…
  - Built-in modules for speed written in C/C++ (e.g. numpy)
  - Several ways to create C/C++ extensions: CPython, **SWIG**…

- **Python for FSW(C/C++) setup**, **desktop execution** & **post-processing**:
  - **Data analysis:** numpy, matplotlib…
  - **Automated regression tests**: py-test
  - **Monte-Carlo** handling



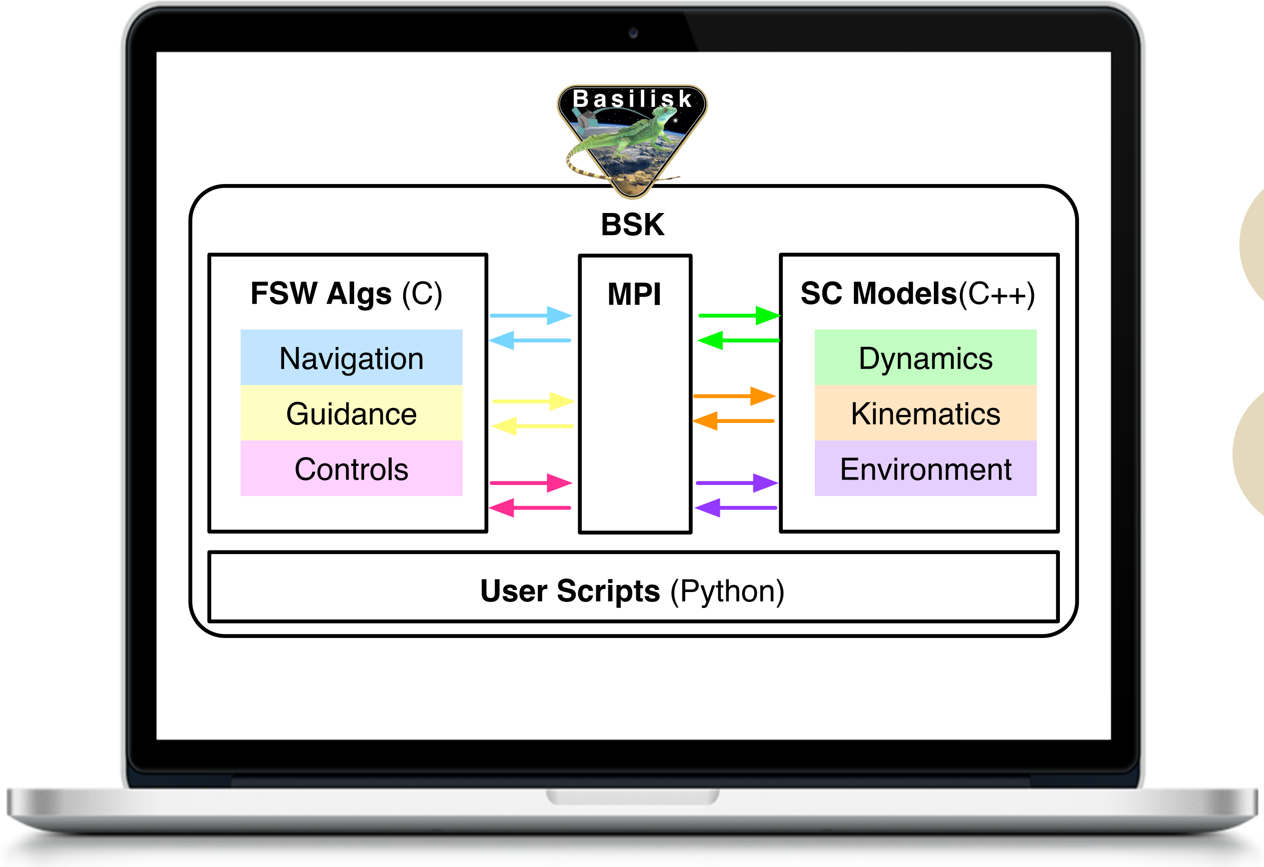
# Basilisk Desktop Testbed: Overview

- **Basilisk:** open-source, cross-platform, desktop testbed for designing flight algorithms and test them in closed-loop dynamics simulations.
- **Language**: C and C++ code wrapped in Python via SWIG
- **AVS & LASP:** interplanetary spacecraft mission support
- **Nominal** (but not required) **Setup:**
  - **Dynamics Process**: simulation of spacecraft physical behavior (C++)
  - **FSW Process**: mission-specific GN&C algorithms (C)
- **Core Elements:**
  - **Hierarchy**: Process -> Task -> Module
  - **Communication**: pub-sub Message Passing Interface

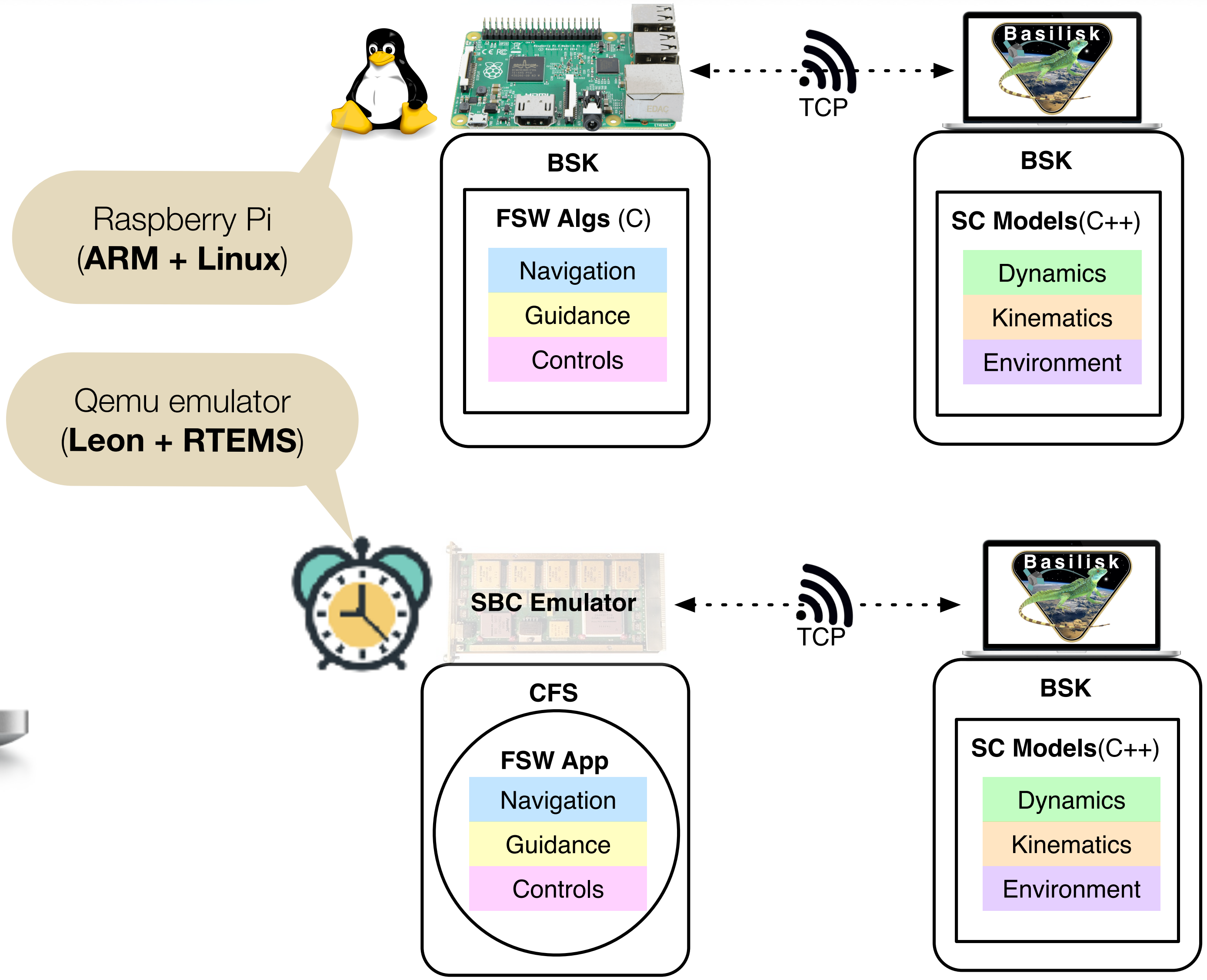


# Migration of the Basilisk Flight Application

- **Single processor environment:**

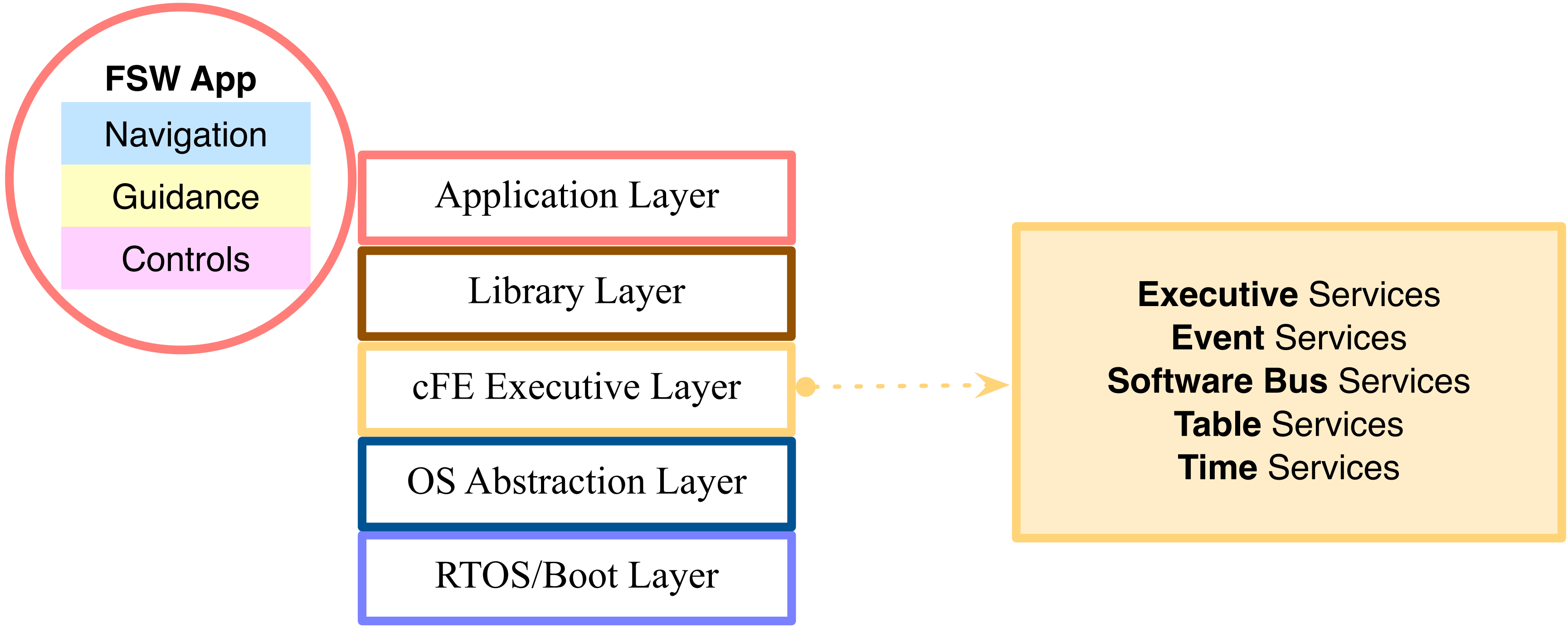



- **Multi processor environment:** realistic testing



# The Core Flight System

- **Middleware layer** (“glue code”) to **ensure portability** among different RTOS and processors.
- **Open-source** product developed by NASA Goddard.
- **Architecture**:

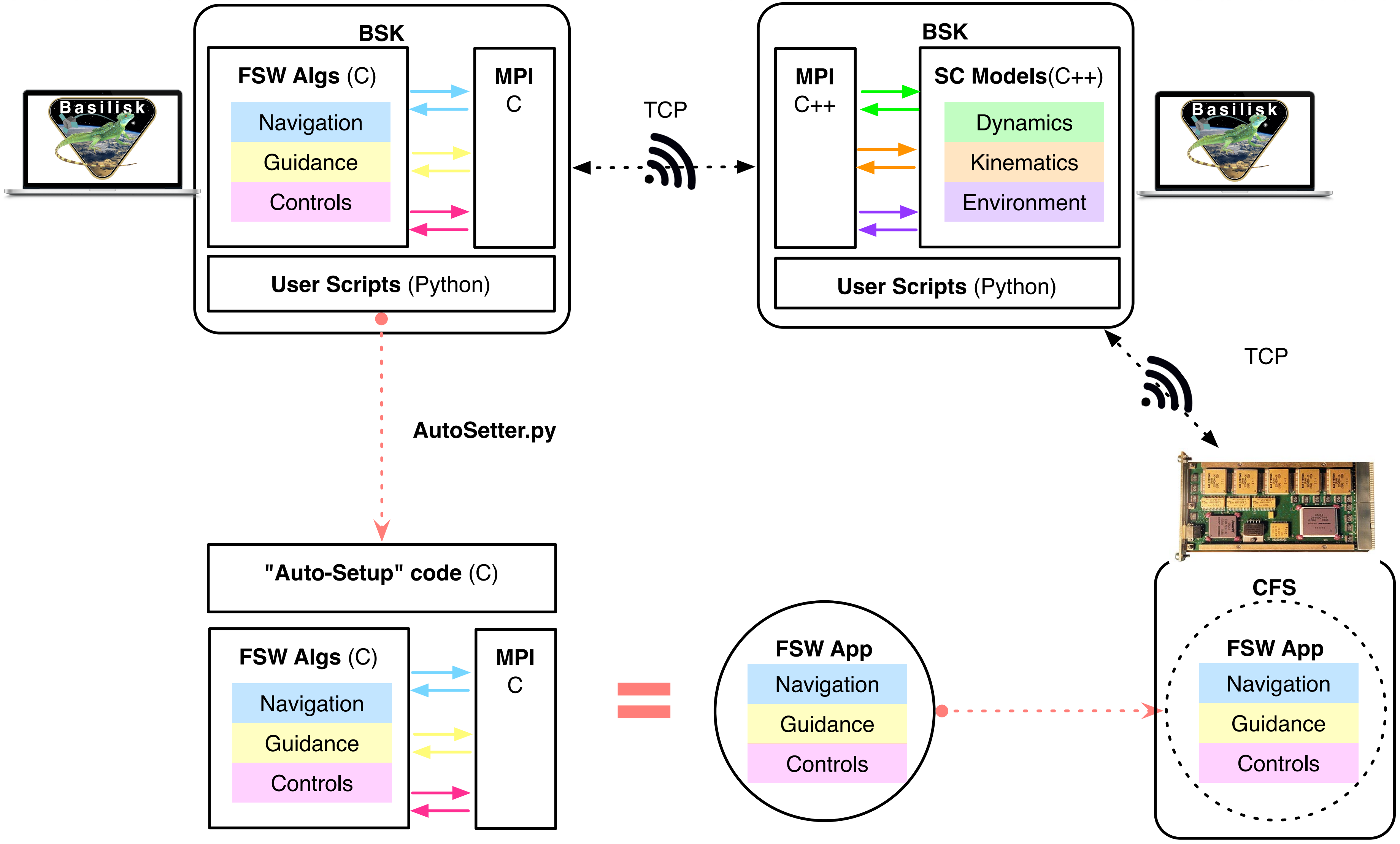


# Basilisk flight algorithms into a CFS application

- **Basilisk C Algorithms + “Auto-Setup” Code:** integrated as a CFS application.

- **Recall desktop dev:** Python for FSW(C/C++) setup, desktop execution and post-processing

- **Python setup:**
  - Initialization of C/C++ modules
  - Grouping of modules in tasks & rates



# Python setup: C module initialization

- **Basilisk C module**: a stand-alone model or self-contained logic.
  - **Config struct**
  - **Generic algorithm calls**: self-init, cross-init, update & reset. [called from Python in desktop exec]
- **Python module initialization**

```c
typedef struct {
    double ISCPntB_B[9];
    double CoM_B[3];
    char outputPropsName[MAX_STAT_MSG_LENGTH];
    int32_t outputPropsID;
}VehConfigInputData;
void SelfInit_vehicleConfigData(VehConfigInputData *ConfigData, uint64_t moduleID);
void CrossInit_vehicleConfigData(VehConfigInputData *ConfigData, uint64_t moduleID);
void Reset_vehicleConfigData(VehConfigInputData *ConfigData, uint64_t callTime, uint64_t moduleID);
void Update_vehicleConfigData(VehConfigInputData *ConfigData, uint64_t callTime, uint64_t moduleID);
```

Vehicle Config Data module

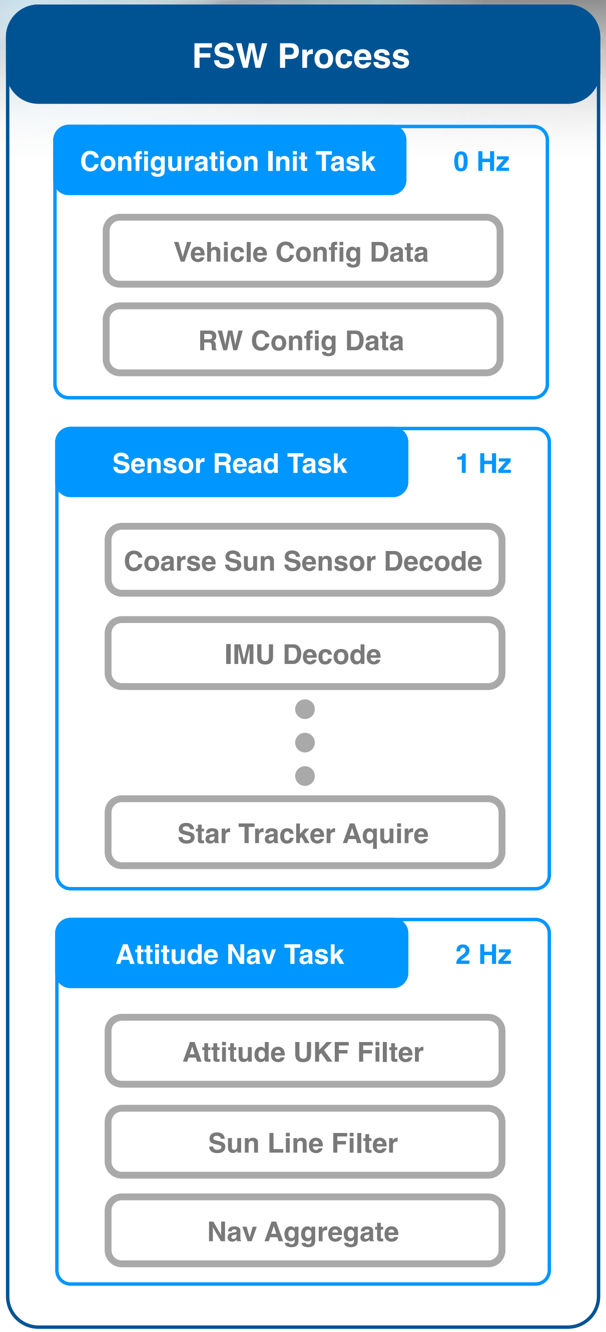
SWIG

```python
        self.VehConfigData = vehicleConfigData.VehConfigInputData()

    def SetVehicleConfigData(self):
        self.VehConfigData.ISCPntB_B = [600.0, 0.0, 0.0, 0.0, 600.0, 0.0, 0.0, 0.0, 600]
        self.VehConfigData.CoM_B = [1.0, 0.0, 0.0]
        self.VehConfigData.outputPropsName = "veh_config_data"
```

# Python setup: task groups & rates

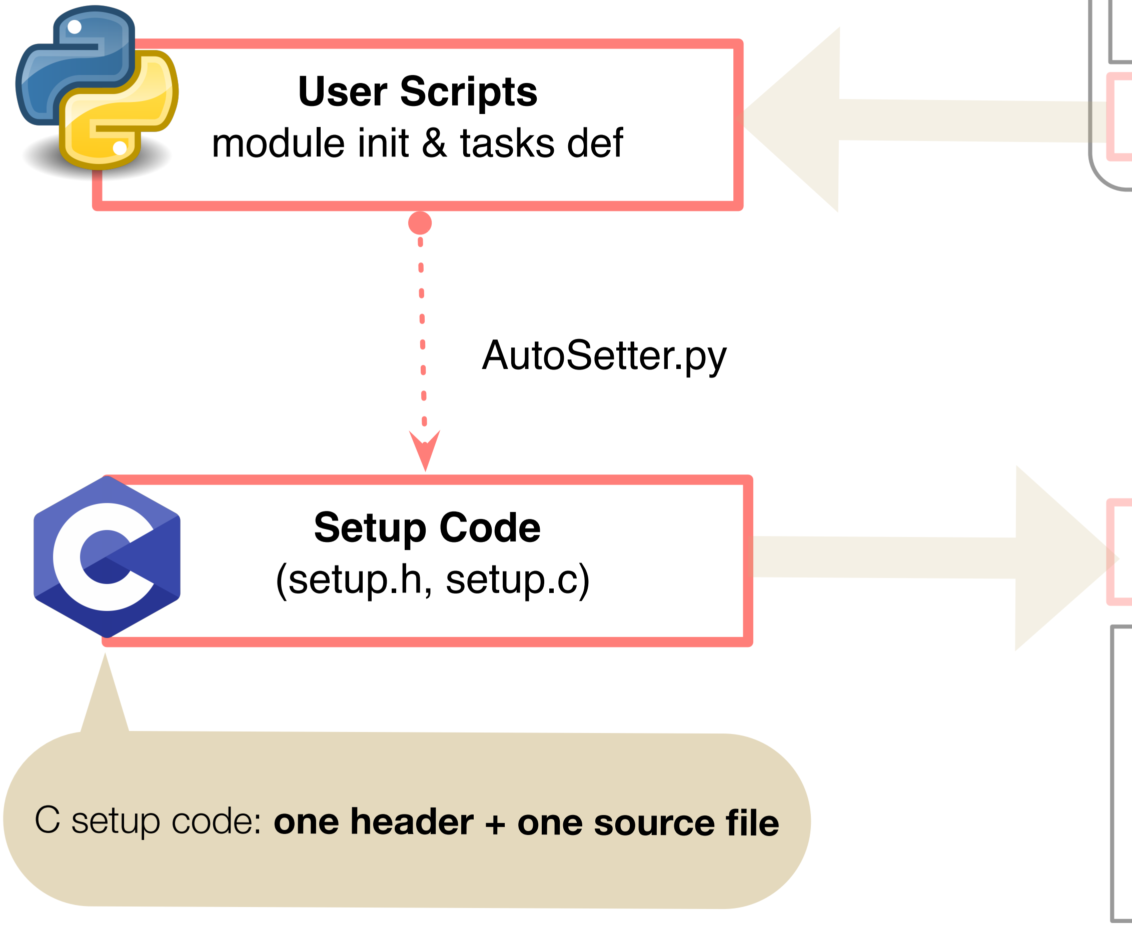
- **Define tasks** at certain rates
- **Add modules to tasks** and define priorities within the task.
- **Examples:**
  - **Config Init Task at 0 Hz:** all modules in the task only called once (at init time)
  - **Sensor Read Task at 1 Hz**: the Update( ) algorithm of each module is called every 1sec, in the priority stablished.

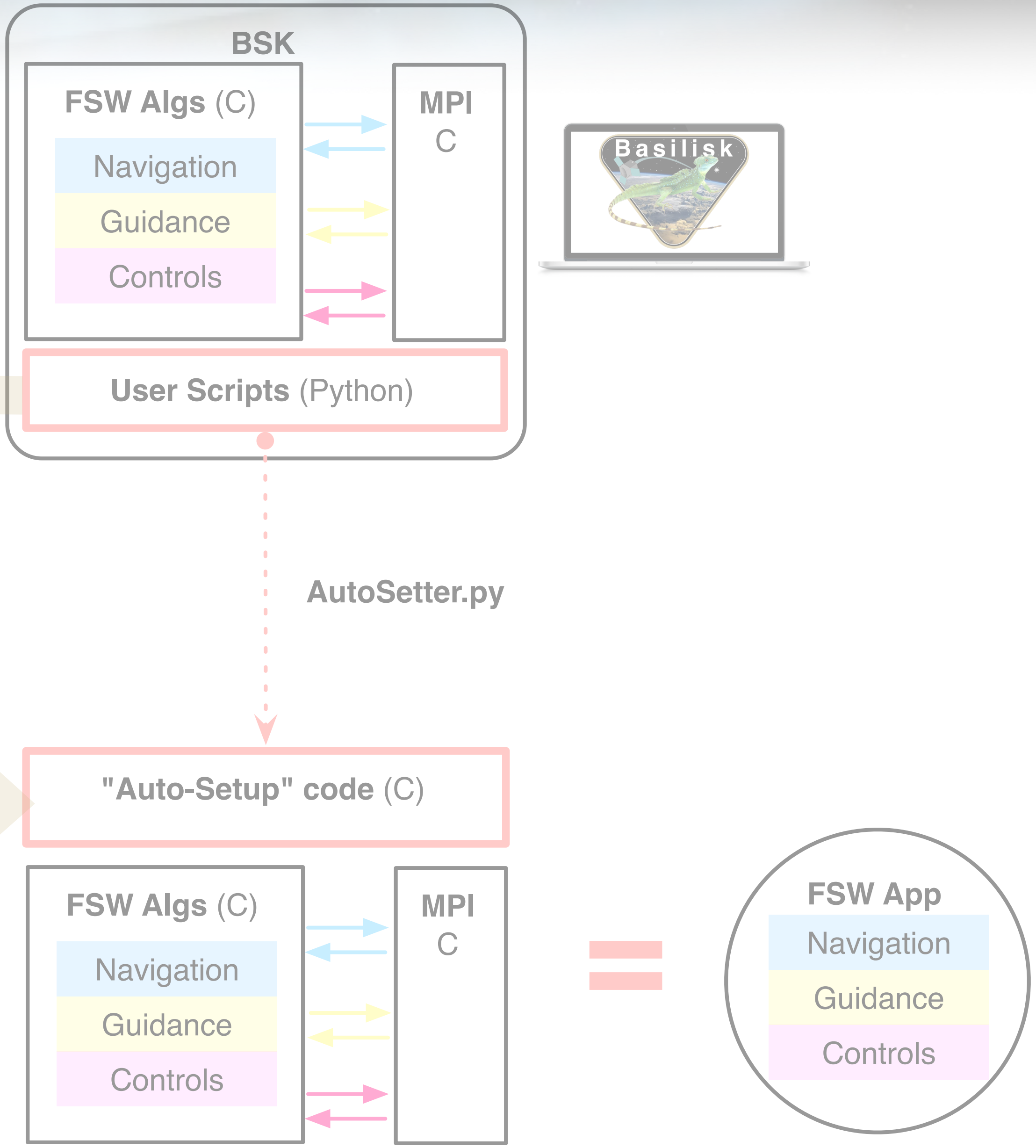
Basilisk hierarchy:
Process —> Tasks —>Modules



# Embeddable FSW Application

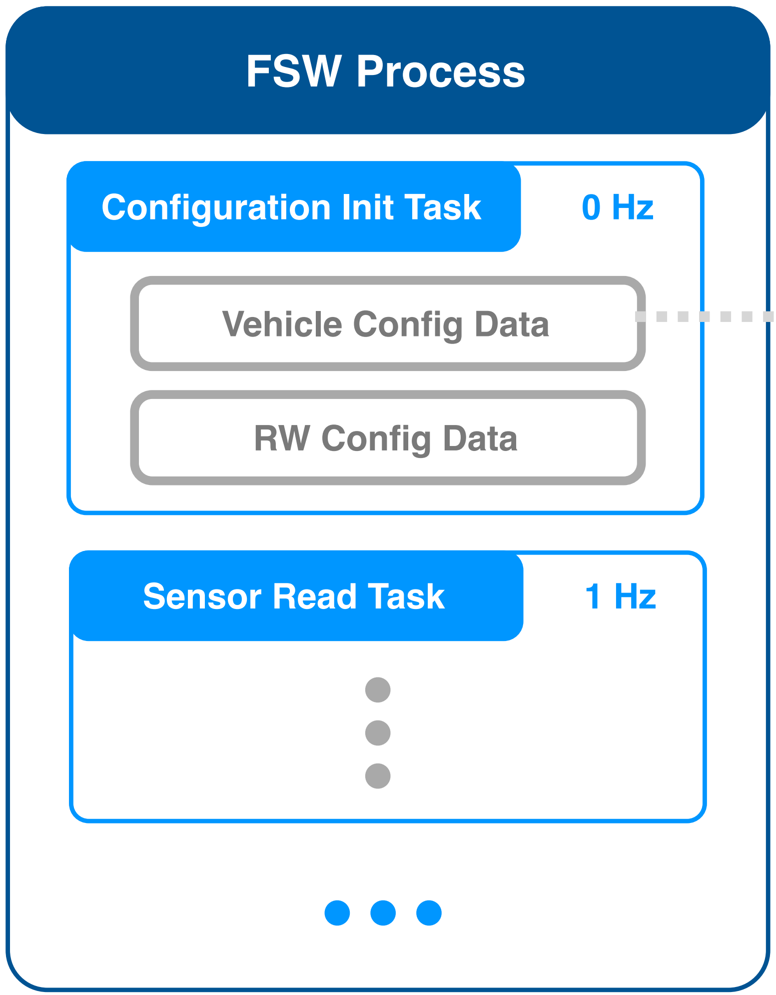
- **Setup code:** needs to be translated from Python to C.
- **FSW algorithm source code**: remains **unchanged**!!

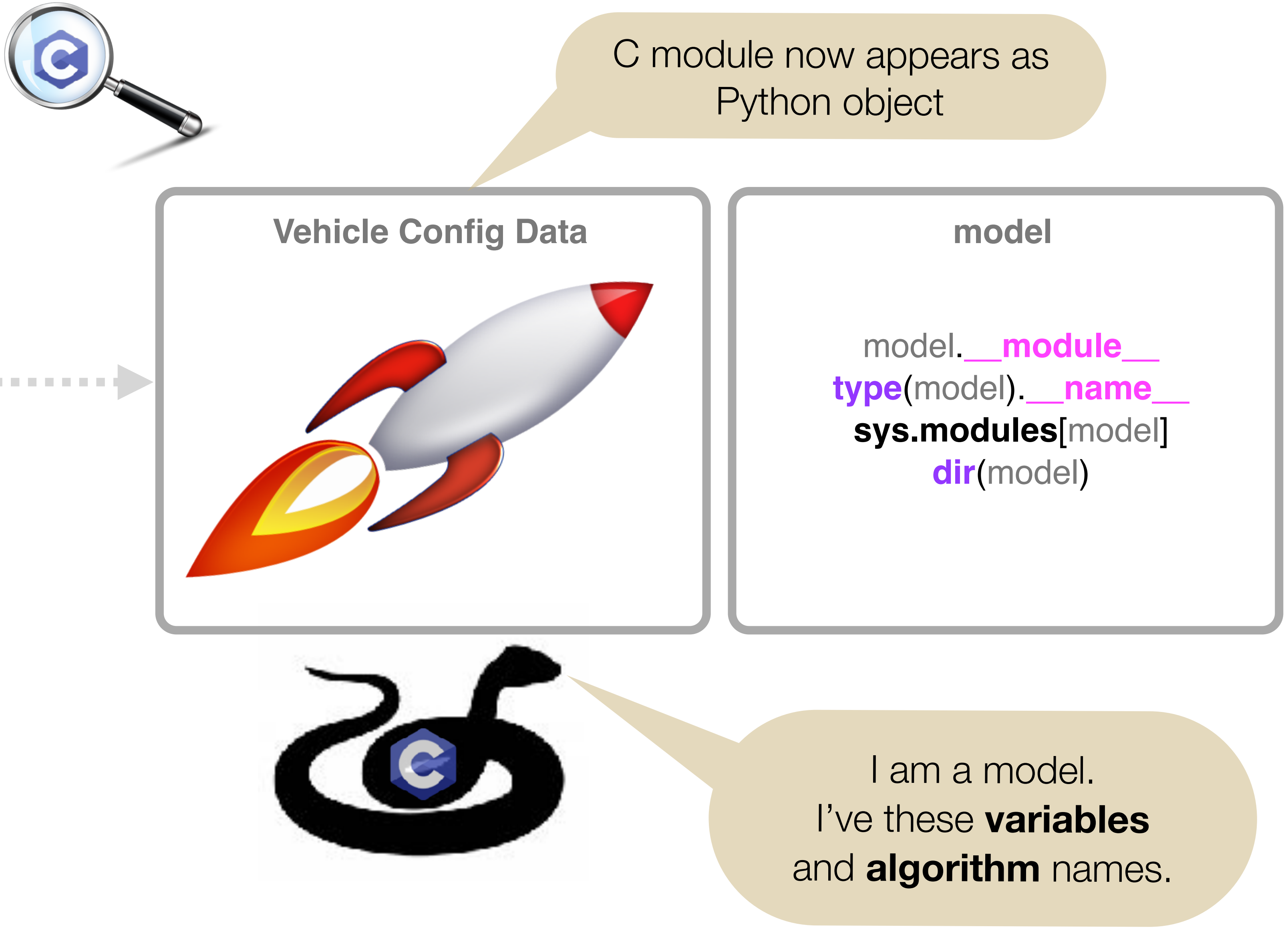


# Python Introspection

- **Why is it so easy** to generate concise C setup code through the “AutoSetter.py”?
- Because Python is great at introspection!



# “AutoSetup.py”: Python input & C output

ISCPntB_B = list of 9 floats
CoM_B = list of 3 floats
outputPropNames = string

```python
    self.VehConfigData = vehicleConfigData.VehConfigInputData()
def SetVehicleConfigData(self):
    self.VehConfigData.ISCPntB_B = [600.0, 0.0, 0.0, 0.0, 600.0, 0.0, 0.0, 0.0, 600]
    self.VehConfigData.CoM_B = [1.0, 0.0, 0.0]
    self.VehConfigData.outputPropsName = "veh_config_data"
```

**Python scenario**
(AutoSetter input)

AutoSetter.py

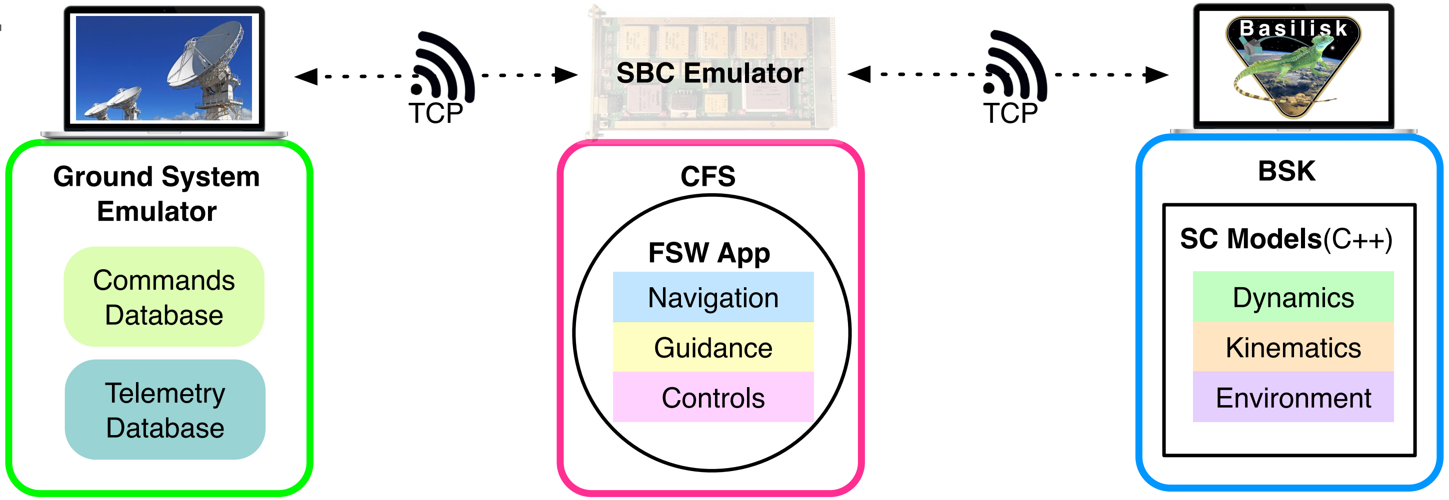
```c
void mPyConfigData_DataInit(mPyConfigData *data){
    memset(data, 0x0, sizeof(mPyConfigData));
    data->initOnlyTask_isActive = 0;

    data->vehConfigData.CoM_B[0] = 1.0;
    data->vehConfigData.ISCPntB_B[0] = 600.0;
    data->vehConfigData.ISCPntB_B[4] = 600.0;
    data->vehConfigData.ISCPntB_B[8] = 600.0;
    strcpy(data->vehConfigData.outputPropsName,"veh_config_data");

    strcpy(data->rwConfigData.rwConstellationInMsgName,"rwa_config_data");
    strcpy(data->rwConfigData.rwParamsOutMsgName,"rwa_config_data_parsed");
    strcpy(data->rwConfigData.vehConfigInMsgName,"veh_config_data");
}

void initOnlyTask_Update(mPyConfigData *data, uint64_t callTime)
{
    Update_vehicleConfigData(&(data->vehConfigData), callTime, 11);
    Update_rwConfigData(&(data->rwConfigData), callTime, 37);
}
```

**C variables init**
(AutoSetter output)

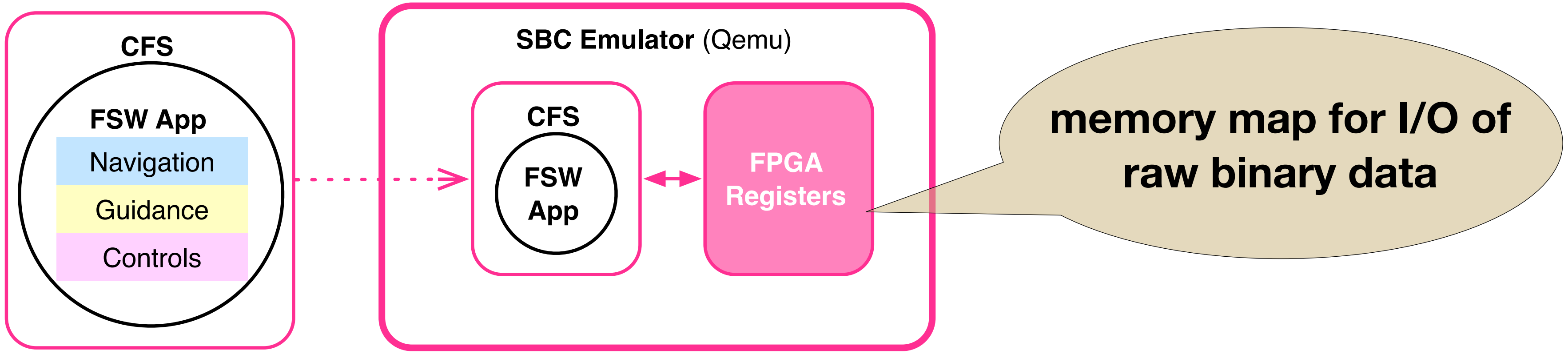
**C algorithm calls**, arranged in tasks
(AutoSetter output)

# Emulated Flat-Sat Configuration

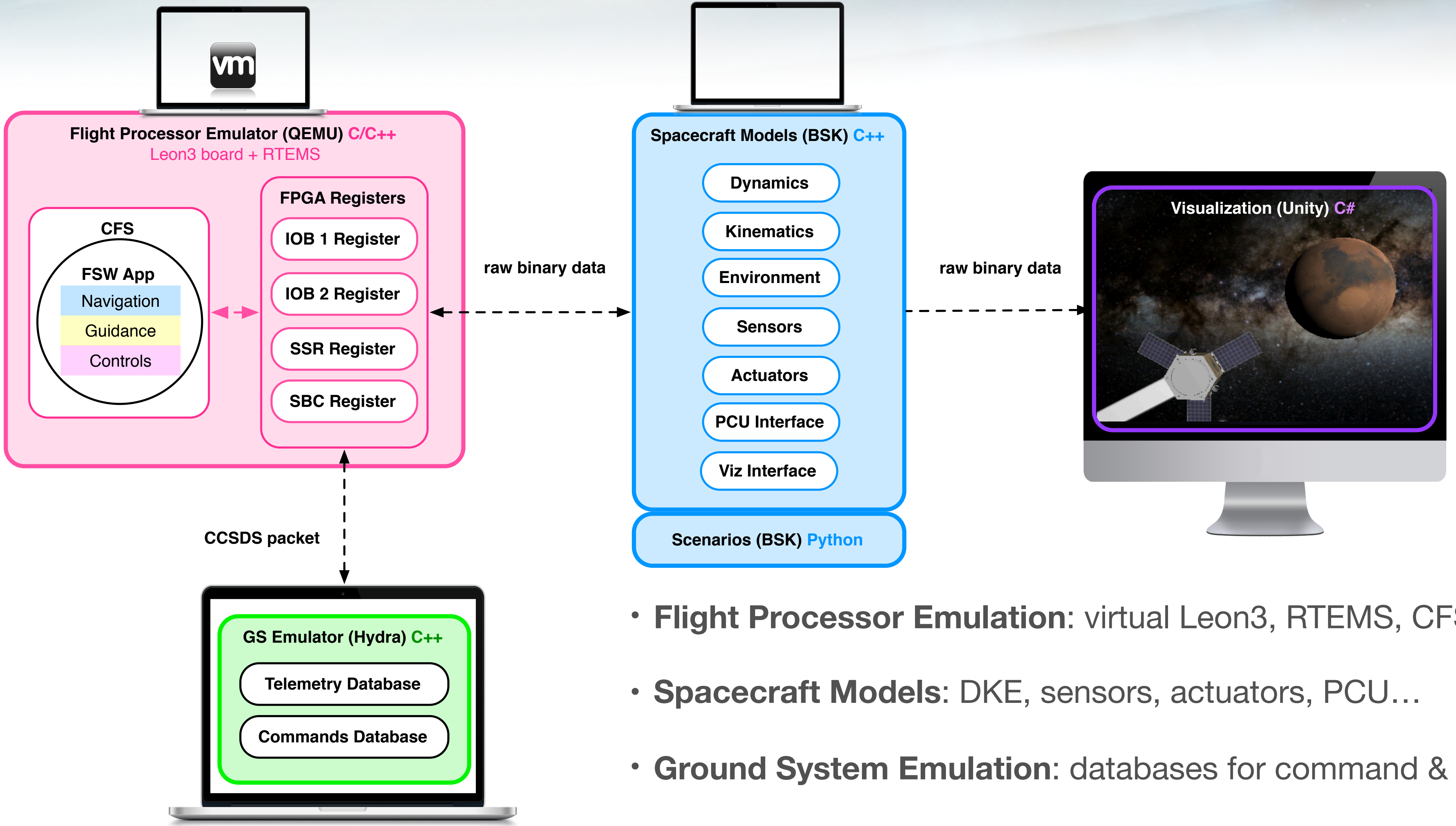
- C flight algorithms + generated C setup code —> integrated as an **embeddable CFS app.**
- **Embedded FSW testing:** closed-loop simulation with other models: s/c physical models, ground system model…




- But interacting with FSW is not that easy when it's embedded… Need to emulate **FPGA Registers**



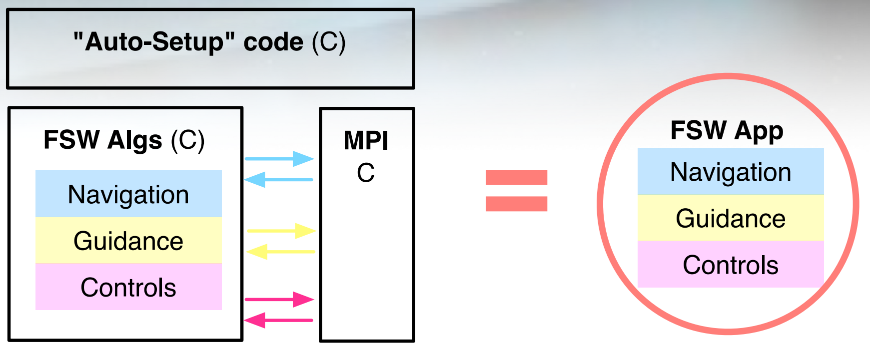
# Emulated Flat-Sat Models

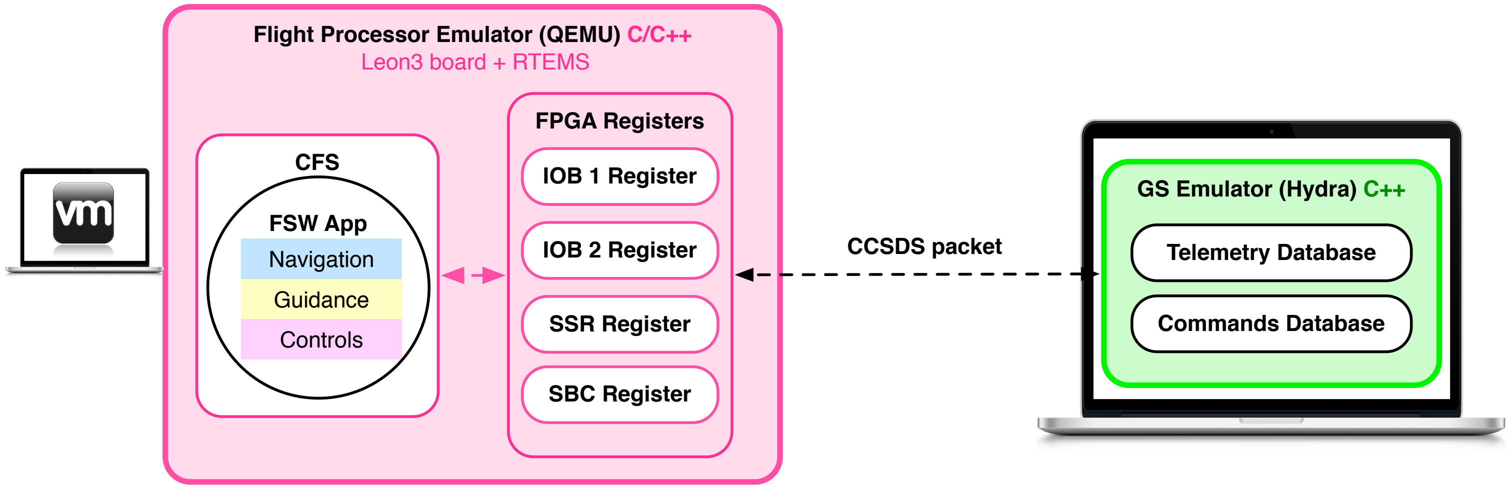


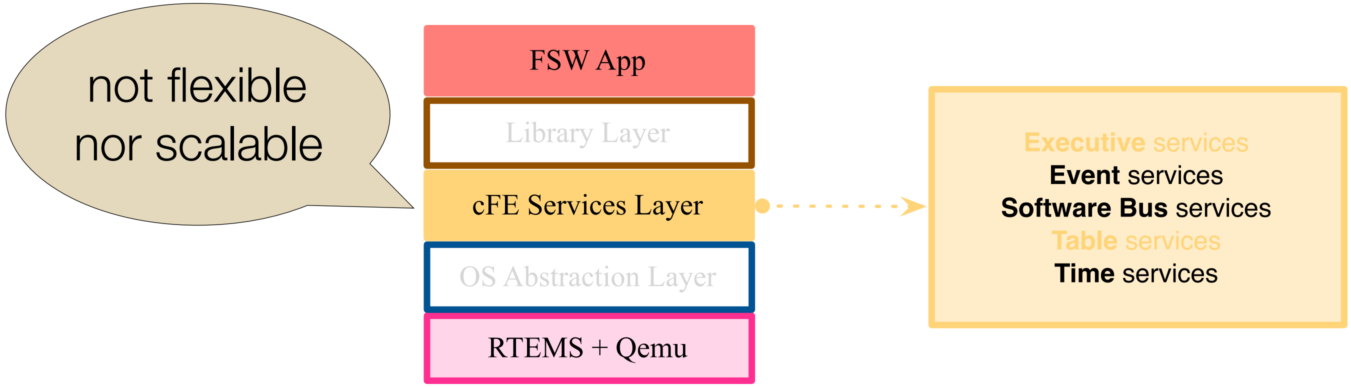

- **Flight Processor Emulation**: virtual Leon3, RTEMS, CFS-FSW App, FPGA reg.
- **Spacecraft Models**: DKE, sensors, actuators, PCU…
- **Ground System Emulation**: databases for command & telemetry
- **Visualization**: Unity GUI

# CFS Embedding Approach: requirements & limitations

- **Does it work?** Yes, and migration is transparent
- **Migration effort**:
  - Generate “Auto-Setup” C code
  - Emulate FPGA registers
- **Difficulties:**
  - Setting flight modes
  - Logging FSW states
- **Replicated CFS functionality:**
  - Software Bus = FSW App’s MPI
  - Time Services = Qemu functionality
  - Event Services = GS functionality



# MicroPython for Embedded FSW Development

- **MicroPython:**
  - Lean and efficient implementation of the Python 3 programming language, **optimized to run in microcontrollers**.
  - **Full of advanced features**: interactive prompt, list comprehension, exception handling… (more alike desktop)
  - Aims to be as **compatible with normal Python** as possible (ease of translation)
- **MicroPython C++ Wrap:**
  - **What?** Header-only C++ library providing interoperability between C/C++ and MicroPython.
  - **Why?** Standard way of extending MicroPython with your own C/C++ modules involves some boilerplate.
- **Python introspection:** for wrapper generation (Same logic as in "AutoSetter.py")
  - Automatically  create a C++ class for every C FSW module
  - Generate MPy integration code-lines for every C++ class

open source initiative®

# MicroPython C++ Wrapping

- C++ class (hpp file) for every C FSW module we have

```
#include "_GeneralModuleFiles/sys_model.h"
#include "vehicleConfigData/vehicleConfigData.h"
class vehConfigDataClass: public SysModel {
public:
    vehConfigDataClass(){ memset(&this->config_data, 0x0, sizeof(VehConfigInputData));}
    ~vehConfigDataClass(){return;}
    void SelfInit(){ SelfInit_vehicleConfigData(&(this->config_data), this->moduleID); }
    void CrossInit(){ CrossInit_vehicleConfigData(&(this->config_data), this->moduleID); }
    void UpdateState(uint64_t callTime){ Update_vehicleConfigData(&(this->config_data), callTime, this->moduleID); }
    void Reset(uint64_t callTime){ Reset_vehicleConfigData(&(this->config_data), callTime, this->moduleID); }

    void Set_outputPropsName(std::string new_outputPropsName){
        memset(this->config_data.outputPropsName, '\0', sizeof(char) * MAX_STAT_MSG_LENGTH);
        strncpy(this->config_data.outputPropsName, new_outputPropsName.c_str(), new_outputPropsName.length());
    }
    std::string Get_outputPropsName() const{
        std::string local_outputPropsName(this->config_data.outputPropsName);
        return(local_outputPropsName);
    }
```

• • •          • • •          • • •

```
private:
    VehConfigInputData config_data;
};
```

C++ class

Callbacks

Setter

Getter

C config struct

# MicroPython C++ Wrapping

- **C++ class for every C FSW module**
- **Generate MPy integration code-lines for every C++ class:** need to register the C++ function and type names so they can be discovered by MicroPython

MPy **function names def**

```
struct vehConfigData_FunctionNames
{
    func_name_def(SelfInit)
    func_name_def(CrossInit)
    func_name_def(Update)
    func_name_def(Reset)
};
```

C++ **class registration**

C++ **function names map**

```
auto mod = upywrap::CreateModule( "fsw" );

upywrap::ClassWrapper <vehConfigDataClass> wrap_vehConfigData("vehConfigDataClass", mod);
wrap_vehConfigData.DefInit <> ();
wrap_vehConfigData.Def < vehConfigData_FunctionNames::SelfInit > (&vehConfigDataClass::SelfInit);
wrap_vehConfigData.Def < vehConfigData_FunctionNames::CrossInit > (&vehConfigDataClass::CrossInit);
wrap_vehConfigData.Def < vehConfigData_FunctionNames::Update > (&vehConfigDataClass::UpdateState);
wrap_vehConfigData.Def < vehConfigData_FunctionNames::Reset > (&vehConfigDataClass::Reset);
wrap_vehConfigData.Property("outputPropsName", &vehConfigDataClass::Set_outputPropsName, &vehConfigDataClass::Get_outputPropsName);
wrap_vehConfigData.Property("ModelTag", &vehConfigDataClass::Set_ModelTag, &vehConfigDataClass::Get_ModelTag);
wrap_vehConfigData.Property("ISCPntB_B", &vehConfigDataClass::Set_ISCPntB_B, &vehConfigDataClass::Get_ISCPntB_B);
wrap_vehConfigData.Property("CoM_B", &vehConfigDataClass::Set_CoM_B, &vehConfigDataClass::Get_CoM_B);
```

**MPy property**: C++ setter & getter

# Desktop Python vs. Embedded MicroPython

Dektop Python script
(C module setup)

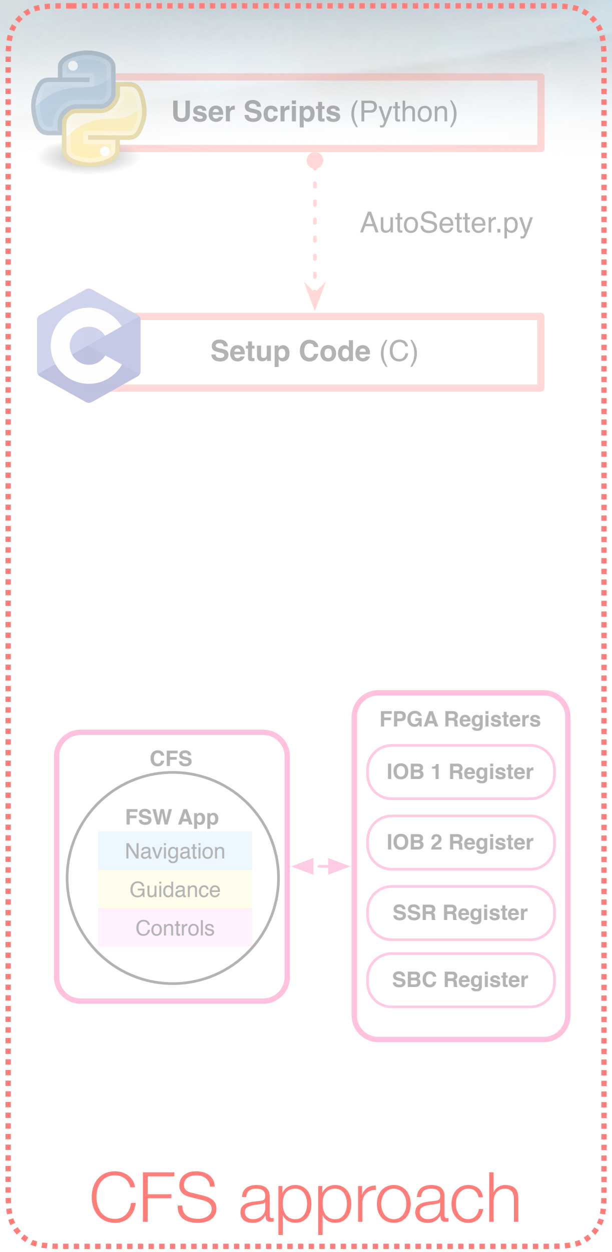
```python
class FSWModels(object):
        def __init__(self, masterSim):
            # Create a sim module as an empty container
            self.simBasePath = masterSim.simBasePath
            # Instantiate C fsw models
            self.VehConfigData = vehicleConfigData.VehConfigInputData()
            self.VehConfigDataWrap = masterSim.setModelDataWrap(self.VehConfigData)
            self.VehConfigDataWrap.ModelTag = "vehConfigData"
            # Initialize models
            self.InitAllFSWObjects()

        def SetVehicleConfigData(self):
            self.VehConfigData.ISCPntB_B = [600.0, 0.0, 0.0, 0.0, 600.0, 0.0, 0.0, 0.0, 600]
            self.VehConfigData.CoM_B = [1.0, 0.0, 0.0]
            self.VehConfigData.outputPropsName = "veh_config_data"
```

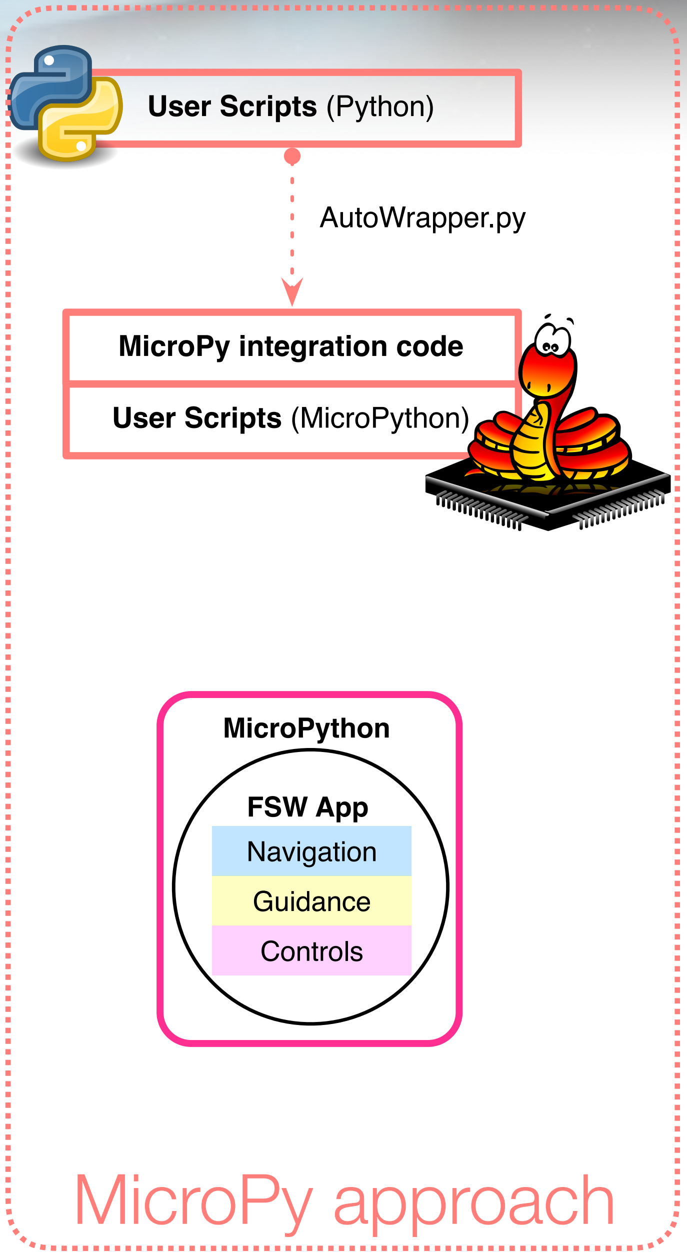
```python
class MPyFSWModels(object):
    def __init__(self, masterSim):
        # Create a sim module as an empty container
        self.simBasePath = masterSim.simBasePath
        # Instantiate cpp classes
        self.VehConfigData = fsw.vehConfigDataClass()
        # Initialize classes
        self.InitAllFSWObjects()

    def SetVehConfigData(self):
        self.VehConfigData.ModelTag = "vehConfigData"
        self.VehConfigData.ISCPntB_B = [600.0, 0.0, 0.0, 0.0, 600.0, 0.0, 0.0, 0.0, 600]
        self.VehConfigData.CoM_B = [1.0, 0.0, 0.0]
        self.VehConfigData.outputPropsName = "veh_config_data"
```

Embedded MicroPy script
(C++ module setup)

# MicroPython Embedding Approach

- **Reduced migration effort:**
  - No more **specific** C setup-code
  - MicroPython integration code is written once (FSW states are **reconfigurable**)
  - No need to emulate FPGA registers
- **Advantages:**
  - Setting flight modes & logging states is easy
  - No more replicated functionality
  - Guaranteed portability



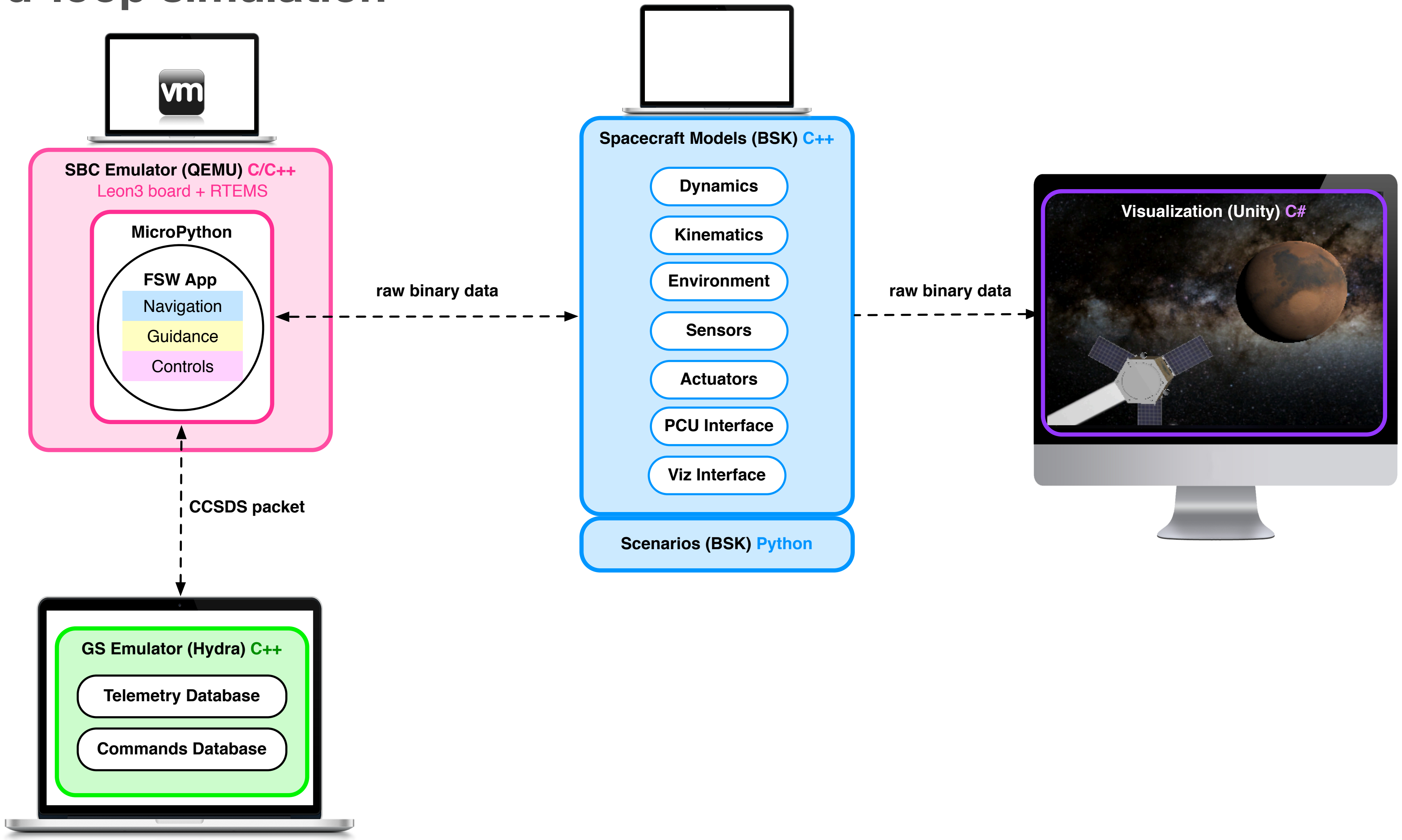



# Future Work

- **Port MicroPython to RTEMS & Leon**

ESA Software Community License – Type 3

- **Distributed closed-loop simulation**



# Thanks for your attention!