Mini Tutorials on COSMOS-core

Eric J. Pilger, Miguel A. Nunes

April 9, 2015

Contents

1 Add a new generic device 2

2 extending the COSMOS namespace 5

2.1 edit nodedef.h 5

2.2 edit jsonlib.c 5

2.3 edit node.ini 6

3 Software profiler 6

3.1 Linux 6

3.2 Mac OS 6

4 Code Documentation using Doxygen 6

5 Installing Latex and Qt with MinGW 6

5.1 LaTex 6

5.2 Custom Qt , MinGW and CMake 6

1 Add a new generic device

As an example we are going to add a new generic device to measure the temperature named “temperatureStation”. Go to jsondef.h approx in line 960 and create the structure that contains the information you want to use.

```
struct temperatureStationStruc
{
        //! Generic info must be here for every device
        genstruc gen;
        //! the following is any data specific to this device
        float temperature; // your temperature data will be stored here
} ;
```

add your temperatureStationStruc structure to the devicestruc union (apporox in line 1400)

```
typedef struct
{
        union
        {
                allstruc all;
                ...
                thststruc thst;
                tsenstruc tsen;
                temperatureStationStruc temperatureStation; // << --- add here
        };
} devicestruc;
```

add your temperatureStationStruc structure to the devspecstruc structure (apporox in line 1500)

```
typedef struct
{
        uint16_t ant_cnt;
        ...
        uint16_t thst_cnt;
        uint16_t tsen_cnt;
        uint16_t temperatureStation_cnt; // << --- add here
        vector<allstruc *>all;
        ...
        vector<thststruc *>thst;
        vector<tsenstruc *>tsen;
        vector<temperatureStationStruc *>temperatureStation;  // << --- add here
} devspecstruc;
```

now go to jsonlib.cpp , add your temperatureStation to the end of device_type_string

```
vector <string> device_type_string
{
        "pload",
        ...
        "cam",
        "temperatureStation" // <-- add here
};
```

in jsondef.h you also must add the device type to the end of device_type enum (approx in line 400)

```
enum
        {
        //! Payload
        DEVICE_TYPE_PLOAD=0,
        ...
        //! Camera
        DEVICE_TYPE_CAM=26,
        //! your tempStation here
        DEVICE_TYPE_TEMPSTATION = 27, // <- add here
        //! List count
        DEVICE_TYPE_COUNT,
        //! Not a Component
        DEVICE_TYPE_NONE=65535
        };
```

now we are going to modify some functions in the code. The first one is json_detroy in jsonlib.cpp

```
void json_destroy(cosmosstruc *cdata)
{
        for (uint16_t i=0; i<2; ++i)
        {
                cdata[i].devspec.ant.resize(0);
                ...
                cdata[i].devspec.tsen.resize(0);
                cdata[i].devspec.temperatureStation.resize(0); // <- add here
                cdata[i].device.resize(0);

        }

        delete [] cdata;
        cdata = NULL;
}
```

(side note: for a really complex type further definitions must be added to the namespace, but most common types are already supported, so this is an advanced feature)

go to json_devices_specific and inside the for loop that goes over each type add some of the following

```
const char *json_devices_specific(string &jstring, cosmosstruc *cdata)
{
...

        for (uint16_t j=0; j<*cnt; ++j)
        {
        ...

                // Dump Temperature Station
                if (!strcmp(device_type_string[i].c_str(),"tempStation"))
                {
                        json_out_1d(jstring,(char *)"device_tempStation_temperature",j,cdata);
                        json_out_character(jstring, '\n');
                }

        }
}
```

go to json_clone

```

int32_t json_clone(cosmosstruc *cdata)
{
...
        case DEVICE_TYPE_TEMPSTATION:
                cdata[1].devspec.tempStation[cdata[1].device[i].all.gen.didx] =
                        &cdata[1].device[i].tempStation;
                break;
...
}
```

add name for the device count in json_addbaseentry

```

uint16_t json_addbaseentry(cosmosstruc *cdata)
{

        ...
        json_addentry("device_tempStation_cnt",
                UINT16_MAX,
                UINT16_MAX,
                offsetof(devspecstruc,tempStation_cnt),
                COSMOS_SIZEOF(uint16_t),
                (uint16_t)JSON_TYPE_UINT16,
                JSON_GROUP_DEVSPEC,
                cdata);

}
```

to json_adddeviceentry add

```

uint16_t json_adddeviceentry(uint16_t i, cosmosstruc *cdata)
{

        ...
        case DEVICE_TYPE_BUS:

                json_addentry("device_tempStation_utc",
                        didx,
                        UINT16_MAX,
                        (ptrdiff_t)offsetof(genstruc,utc)+i*sizeof(devicestruc),
                        COSMOS_SIZEOF(double),
                        (uint16_t)JSON_TYPE_DOUBLE,
                        JSON_GROUP_DEVICE,
                        cdata);

                json_addentry("device_tempStation_temperature",
                        didx,
                        UINT16_MAX,
                        (ptrdiff_t)offsetof(temperatureStationStruc,temperature) +
                                i*sizeof(devicestruc),
                        COSMOS_SIZEOF(double),
                        (uint16_t)JSON_TYPE_DOUBLE,
                        JSON_GROUP_DEVICE,
                        cdata);

                cdata[0].devspec.tempStation.push_back(
                        (temperatureStationStruct *)&cdata[0].device[i].tempStation);
                cdata[0].devspec.tempStation_cnt =
                        (uint16_t)cdata[0].devspec.tempStation.size();
        break;

}
```

2 extending the COSMOS namespace

example on extending the COSMOS namespace by adding port/address name to the devices namespace.

2.1 edit nodedef.h

look for nodestruc_s
look for imustruc_s, line 220
add connection information for device:

```
char port[COSMOS_MAX_NAME];
```

note: ”COSMOS_MAX_NAME” contains 40 spaces
add this to other devices that may require the name information:

- imu
- stt
- rw
- tcu (mtr)
- gps
- payload
- cpu

now recompile and check everything works

2.2 edit jsonlib.c

add the following JSON strings to the COSMOS namespace table

```
("cpu_port",JSON_\TYPE_\STRING)
("gps_port",JSON_\TYPE_\STRING)
("imu_port",JSON_\TYPE_\STRING)
("stt_port",JSON_\TYPE_\STRING)
("mtr_port",JSON_\TYPE_\STRING)
("payload_port",JSON_\TYPE_\STRING)
("rw_port",JSON_\TYPE_\STRING)
```

in jason_setup(), line 2600, add entries by copying a static one:

```
    json_addentry("cpu_port",i,-1,
    offsetof(cosmosstruc,stat.node.stt)+
    (ptrdiff_t)offsetof(cpustruc_s,algn)+i*sizeof(cpustruc_s));
```

do the same to the: imu, stt, gps, mtr, rw, payload.

2.3 edit node.ini

add ”imu_port”,”/dev/ttyUSB0”
note: later this will be done automatically using the COSMOS editor that Kyle is working on.
to finally use the port name, ex.: microstrain_connect(cosmos_data.stat.imu[0].port)

—————-

things to clarify
COSMOS namespace vs C++ namespace

notes: node.ini is a static description of the node
all the node.ini information goes into the node_s structure
node_s - description info about node node_d - dynamic part of node, like telemetry

3 Software profiler

3.1 Linux

To check how your software preforms in Linux use 'gprof'

1) Compile with correct switches -pg

CFLAGS = -pg

go to examples/profiler $ make testprofiler

2) run to completion, exit normally this will create file gmon.out

3) gprof ¡program¿ it reads gmon.out and prints a report

3.2 Mac OS

Use Instruments budled with Xcode or install http://valgrind.org

Here is a list of profiling tools recomended by Qt:
`http://qt-project.org/wiki/Profiling-and-Memory-Checking-Tools`

4 Code Documentation using Doxygen

Download Doxygen from www.doxygen.org

5 Installing Latex and Qt with MinGW

5.1 LaTex

Windows only:

Download Latex from Here: `http://www.tug.org/protext/`

Protext will contain MiKTex and TeXstudio.

-extract files to a convenient folder

-Install folder has a pdf of installation procedures

-follow that document to get TeXstudio up and running. TeXstudio allows you to edit .tex files.

5.2 Custom Qt , MinGW and CMake

Windows only:

Download Qt4.8.5.zip, and the cmake, mingw and qt-5.2.0 executables from here: `http://cosmos-project.org/software/downloads/`

- Run the Qt5 executable and an installation wizard will guide you through installation. Accept all defaults.
- Run the cmake executable, accepting all defaults.
- Run the mingw-builds executable, accepting all defaults.
- Expand the Qt4.8.5 zip and place it in the Qt folder created on the C: drive by the Qt5 install.

When you first run Qt Creator, you will need to go to the Build & Run section of Tools:Options to set up your environment. You will need to point to CMake, Compilers, and Qt Versions. Browse, or add then browse, as appropriate. The paths should be something like:

- CMake: C:\Program Files (x86)\CMake 2.8\bin\cmake.exe
- Compilers: C:\Program Files (x86)\mingw-builds\x32-4.8.1-posix-dwarf-rev5\mingw32\bin\g++.exe
- Qt Versions: C:\Qt\Qt4.8.5\bin\qmake.exe

Once this has been established, go to the Kits tab and set up a custom kit by selecting the things you just added.